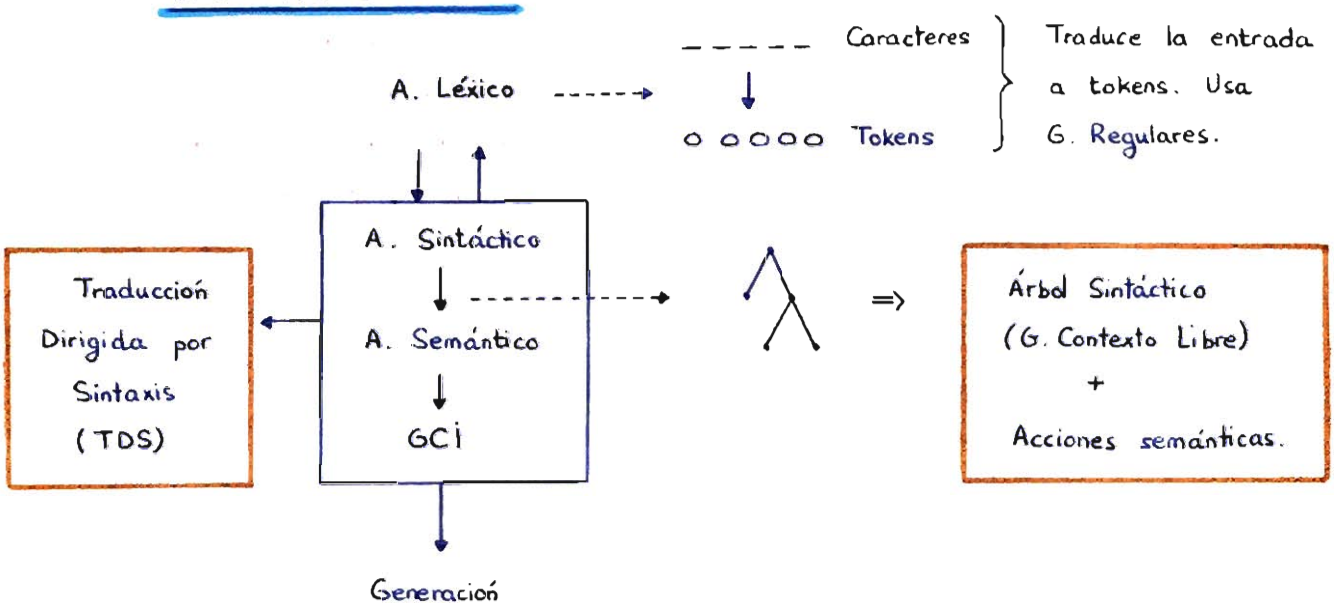
TEMA 7: ANÁLISIS SEMÁNTICO Y GENERACIÓN DE CÓDIGO INTERMEDIO.

1. INTRODUCCIÓN.



En la Traducción Dirigida por Sintaxis se asocia cierta información al lenguaje. Así, tendremos la gramática de contexto libre (GCL) y además unas REGLAS SEMÁNTICAS o ACCIONES.

ACCIONES (REGLAS SEMÁNTICAS) = ATRIBUTOS + REGLAS

Los ATRIBUTOS son el mecanismo que permite manejar la información semántica asociada a los símbolos de la gramática. Esta información asociada son los tipos de la gramática. Tendremos un atributo por cada información semántica que se necesite manejar de cada símbolo.

Ejemplo:

Regla: $A \to XYZ$, $A \to id := E$

Atributo asociado a A → A.a
Atributo asociado a X → X.p
Atributo asociado a Y → Y.b

Posible regla semántica → A.a := X.p + Y.b

Las REGLAS SEMÁNTICAS permiten calcular el valor de los atributos asociados a un símbolo utilizando los valores de otros símbolos de esa misma regla sintáctica.

Ejemplo:

Regla sintáctica: $A \rightarrow id := E$

Atributos: id.tipo
E.tipo
A.tipo

No se puede escribir una regla semántica con un atributo de un símbolo que no pertenezca a la regla sintáctica asociada.

Regla semántica:
```
A.tipo := if id.tipo = E.tipo then
            OK_tipo;
          else
            error_tipo;
```

MAL

Regla semántica: A.tipo := if id.tipo = E.tipo + C.tipo ...

→ Esta regla es incorrecta porque C no aparece en la regla sintáctica y por tanto carezco de información sobre sus atributos

Al conjunto de GCL + Reglas semánticas (atributos y reglas) se le conoce como Traducción Dirigida por Sintaxis (TDS).

La TDS consiste en asociar a cada regla sintáctica una regla semántica que permite hacer la comprobación de tipos y la generación de código intermedio.

1.	$S \rightarrow AB$	$\Rightarrow$	1.	$A.a = f(B.b, S.b)$
2.	$A \rightarrow Bb$	$\Rightarrow$	2.	
3.	$A \rightarrow c$	$\Rightarrow$	3.	⋮
4.	$B \rightarrow fA$	$\Rightarrow$	4.	
	Regla sintáctica			Regla semántica

El analizador semántico se encarga de la comprobación de tipos. De hecho, el analizador semántico casi va a quedar reducido a dicha función.

La unicidad de las variables se comprobaba ya en el léxico con las acciones semánticas.

2. TRADUCCIÓN DIRIGIDA POR SINTAXIS (TDS).

La TDS es la técnica que se utiliza tanto para hacer la comprobación de tipos como para construir el generador de código intermedio. Consiste en definir atributos para los elementos gramaticales de la GCL (gramática de contexto libre) y asociar reglas semánticas a las reglas sintácticas, de manera que permitan calcular el valor de un atributo en función del valor de los atributos de los símbolos de la regla. A cada regla sintáctica se le asocia una regla semántica.

2.1. NOTACIONES.

Hay 2 posibles notaciones para expresar la TDS:

1) Definición Dirigida por Sintaxis (DDS):

Alto nivel. No se dan detalles, únicamente se indica la regla semántica correspondiente a cada regla sintáctica, sin decir en qué momento se aplica la regla semántica.

Ejemplo:

(Regla sintáctica)		(Regla semántica)
$T \rightarrow int$	$\longleftrightarrow$	$\{T.tipo := entero\}$

2) Esquema de traducción (EDT):

Bajo nivel. Las acciones (reglas semánticas) se intercalan entre los consecuentes de las reglas. Para ello se ponen entre { } dichas acciones.

Ejemplo:

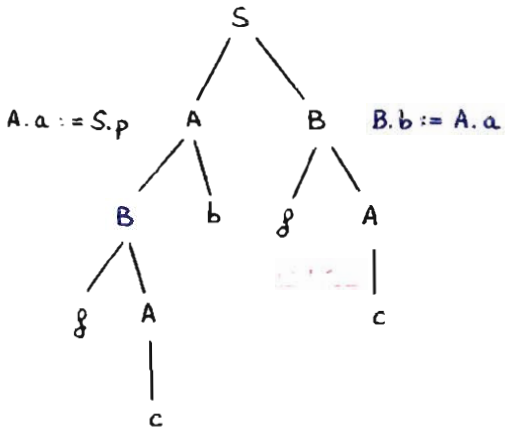
* Regla: $S \rightarrow AB$

- Representación DDS: $1.\ S \rightarrow AB \Longrightarrow 1.\ \{ A.a := S.p\ ,\ B.b := A.a \}$

- Representación EDT: $1.\ S \rightarrow \{A.a := S.p\}\ A\ \{B.b := A.a\}\ B$

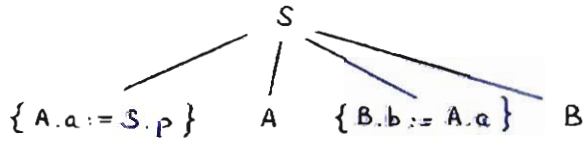
* Reglas:
1. $S \rightarrow AB$
2. $A \rightarrow Bb$
3. $A \rightarrow c$
4. $B \rightarrow gA$

- Representación DDS: $1.\ \{ A.a := S.p\ ,\ B.b := A.a \}$



ÁRBOL ANOTADO DDS:
En cada nodo del árbol anotado recogemos los valores de los atributos asociados al símbolo del nodo.

- Representación EDT: $1.\ S \rightarrow \{ A.a := S.p\}\ A\ \{B.b := A.a\}\ B$



ÁRBOL ANOTADO EDT:
Las acciones semánticas {} forman parte del árbol anotado. Se conoce el orden de ejecución.

2.2. ATRIBUTOS.

Existen 2 tipos de atributos:

- **Sintetizados**: Un atributo es sintetizado si su valor en un nodo del árbol se calcula como una función de valores de atributos de hijos de ese nodo.

- **Heredados**: (Todos los que no son sintetizados). Un atributo es heredado si su valor en un nodo del árbol se calcula como una función de valores de atributos del padre o hermanos de ese nodo.

Ejemplo:

- Regla: $A \to XYZ$ Atributo: $A.s := f(X.x, Y.y)$
  ó
  $A.s := g(X.x, Z.z)$

El atributo A.s es sintetizado en ambos casos porque se calcula en función de los hijos de A (A tiene tres hijos: X, Y, Z), es decir, se calcula en función del consecuente de la regla.

- Reglas: $A \to XYZ$, $X \to Bb$ Atributo: $X.x := f(B.b)$

El atributo X.x es sintetizado: sólo depende de los hijos.

- Reglas: $A \to XYZ$, $X \to Bb$ Atributo: $X.h := f(A.s)$

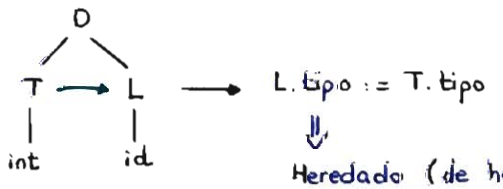
El atributo X.h es heredado: depende de su padre.

- Reglas: $A \to XYZ$, $X \to Bb$ Atributo: $Z.h := f(A.s, X.x)$

El atributo Z.h es heredado: depende de su padre y hermano.

Ejemplo:

Reglas: $D \to TL$, $T \to$ int | float, $L \to$ id



(78)

* SIMULTANEAR SINTÁCTICO Y SEMÁNTICO:

Al hacer el diseño del traductor tenemos dos posibilidades:

- Simultanear el analizador sintáctico y el semántico, ir haciéndo ambos análisis a la vez.
- No simultanearlos, por lo que se hace primero el analizador sintáctico, luego el analizador semántico y, por último, la generación de código intermedio.

Que se pueda o no simultanear va a depender de una serie de normas que se deben cumplir. Es más cómodo que se puedan simultanear, pero no es un error que no se pueda hacer al mismo tiempo.

Ejemplo:

Regla: $A \rightarrow XYZ$

* Atributo $X.h = f(A.a)$ ⟶ Sí se puede simultanear el a. sintáctico con el a. semántico.

* Atributo $X.h = f(A.a, Y.y)$ → No se puede simultanear el a. sintáctico con el a. semántico.

* Atributo $Y.h = X.h$ ⟶ Sí se puede simultanear.

* Atributo $Y.h = g(X.h, Z.z)$ → No se pueden simultanear.

Como vemos, el a. sintáctico y el a. semántico se pueden simultanear cuando se utilizan atributos que se calculan en función de atributos que fluyen de izquierda a derecha. Cuando se utilizan atributos de la derecha estamos obligando a hacer primero el sintáctico y luego el semántico.

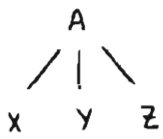
Para poder simultanear a. sintáctico y a. semántico la información debe fluir de izquierda a derecha: usar ATRIBUTOS SINTETIZADOS y ATRIBUTOS HEREDADOS POR LA IZQUIERDA, heredados por la derecha no.

Sea $A \rightarrow X_1 X_2 \dots X_n$

Cada atributo heredado por la izquierda de X_j $(1 \leq j \leq n)$ depende sólo de:

- Los atributos de los símbolos $X_1 X_2 \dots X_{j-1}$ (los situados a la izquierda de X_j en el consecuente) => Heredados de hermanos.
- Los atributos de A => Heredados del padre.

Ejemplo:



- Para calcular atributos de A puedo utilizar información de los atributos de X, Y, Z.
- Para calcular atributos de X puedo utilizar información de los atributos de A.
- Para calcular atributos de Y puedo utilizar información de los atributos de A, X.
- Para calcular atributos de Z puedo utilizar información de los atributos de A, X, Y.

En el diseño puedo introducir todos los atributos que necesite.
¿Qué pasa con el lexema de las variables? Tengo 2 posibilidades:

1) El analizador léxico introduce el lexema en la Tabla de Símbolos.
   En este caso la función sería:
   completa (idl.posición, T.tipo, despl)
   Posición que ocupa en la TS.
2) El analizador léxico no introduce el lexema en la Tabla de Símbolos.
   En este caso la función sería:
   completa (idl.lexema, T.tipo, despl)
   Hay que comprobar repeticiones y luego buscarle una posición libre.

Ejemplo 1:

Gramática de Contexto Libre:

1. P → D
2. D → D ; D
3. D → id : T
4. T → integer
5. T → real

Con esta gramática sólo se reconoce la declaración de variables.

Código:

```
x : integer ;
y : real ;
z : real ;
```

DESPLAZAMIENTO:
- Integer → 4
- Real → 8

Tabla de símbolos:

Lexema	Tipo	Desplaz.
x	integer	0 (+4)
y	real	4 (+8)
z	real	12 (+8)
R	integer	20 (+4)

⋮

El campo desplazamiento me permite conocer la dirección de memoria relativa de las variables. El campo tipo y el campo desplazamiento los tengo que rellenar en función de las acciones que realice el analizador semántico

En cada nuevo bloque (nuevo ámbito de variables) se utiliza una nueva TS y las variables comienzan con desplazamiento 0.

Utilizo notación DDS (Definición Dirigida por Sintaxis):

1. P → D	1. despl = 0 ;
2. D → D ; D	
3. D → id T	3. completar (id, T.tipo, despl) ; despl := despl + T.despl ;
4. T → integer	4. T.tipo := integer ; T.despl := 4 ;
5. T → real	5. T.tipo := real ; T.despl := 8 ;

Añadir que el identificador es del mismo tipo que T y luego insertarlo en la TS.

Ejemplo 2:

Gramática de Contexto Libre:

1. P → D
2. D → D ; D
3. D → id : T
4. T → integer
5. T → real
6. T → array [num] of T_1

Utilizo notación EDT (Esquema De Traducción):

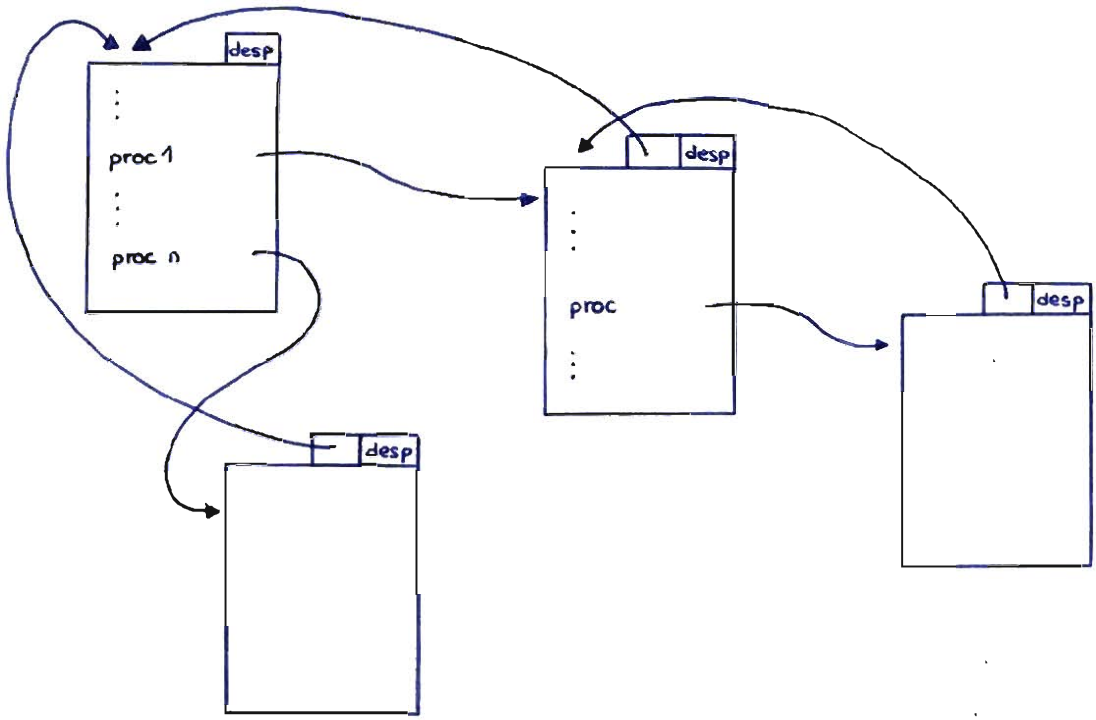
1. P → { despl = 0 } D
2. D → D ; D
3. D → id : T { completar (id.lex , T.tipo, despl) , despl := despl + T.ancho }
4. T → integer { T.tipo := entero , T_{ancho} := 4 }
5. T → real { T.tipo := real, T_{ancho} := 8 }
6. T → array [num] of T_1 { T.tipo := array , T_{ancho} := T_1.ancho * num.valor }

Ejemplo 3: Hacer el EDT (Esquema De Traducción) de la siguiente Gramática de Contexto Libre (GCL):

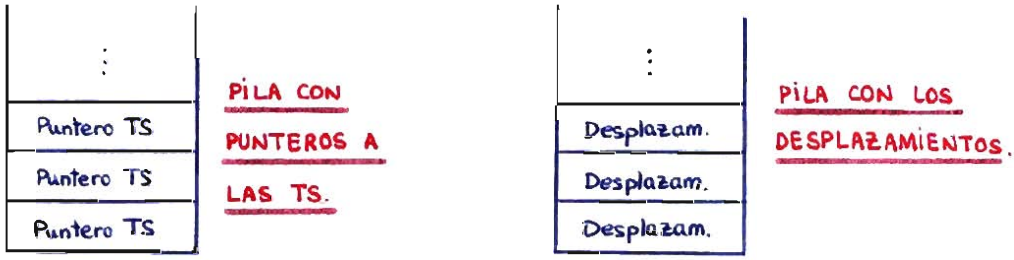
1. P → D
2. D → D ; D
3. D → proc id ; D ; S
4. D → id : T
5. T → integer
6. T → real
7. S → ... sentencias ...

Con esta gramática se pueden declarar variables externas al procedimiento y luego un procedimiento con sus correspondientes variables.

Voy a tener varios bloques (ámbitos), lo que me lleva a tener diferentes Tablas de Símbolos. En la cabecera de cada una de las tablas voy a tener un puntero a la TS anterior y un campo desplazamiento con el ancho total de la TS en cuestión (variables declaradas). Este campo no es necesario siempre.



Además de las TS tendremos 2 pilas: una pila con punteros a las Tablas de Símbolos y otra pila con desplazamientos.



En la cima de la pila de punteros tendremos el puntero a la TS que está actualmente activa. Al acabar con una TS extraemos su puntero (la cima de la pila) y el puntero de la nueva cima será la nueva TS activa.

En la cima de la pila de desplazamientos tendremos el desplazamiento total de la TS activa. El desplazamiento anterior en la pila es el desplazamiento de la tabla padre (TS anteriormente activa).

1. P → D
2. D → D ; D
3. D → proc id ; D ; S -- Declaración de procedimiento.
4. D → id : T -- Declaración de variable.
5. T → integer
6. T → real
7. S → ... sentencias ...

Acciones semánticas:

⑤ { T.tipo := integer , T.ancho := 4 }

⑥ { T.tipo := real , T.ancho := 8 }

④ { introducir (cima (punt_TS) , id , T.tipo , T.ancho) ,
cima (despl) := cima (despl) + T.ancho }

Cuando declaro una variable necesito conocer:

- Tabla de Símbolos activa.
- El lexema o la posición del identificador.
- El tipo.
- El ancho → Con este dato recalculo el nuevo desplazamiento total de la TS: obtengo el desplazamiento de la TS actual, le sumo el ancho de la variable que se acaba de declarar, y finalmente lo guardo de nuevo en la pila de desplazamientos.

③ D → proc id ; { t := crear_tabla (cima (punt_TS)) ,
introduce_proc (cima (punt_TS) , id.lexema , t) ,
push (t , punt_TS) ,
push (0 , despl) }
D ; S { añade_ancho (cima (punt_TS) , cima (despl)) ,
pop (punt_TS) ,
pop (despl) }

Cuando creo un nuevo procedimiento:

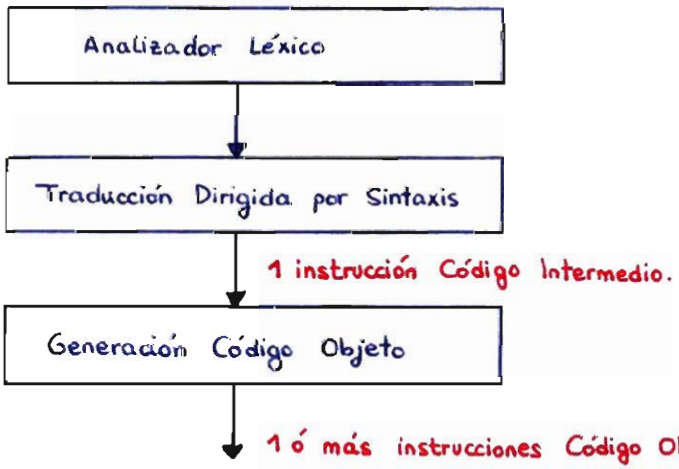
- Creo su TS y hago que apunte a su tabla padre.
- Introduzco en la tabla activa el lexema del procedimiento nuevo y el puntero a su TS (la que acabo de crear).
- Introduzco en las respectivas pilas el desplazamiento nuevo (inicialmente 0) y el puntero de la TS nueva (es decir, activo la TS del procedimiento).

① P → { t := crear_tabla_padre (NULL),
push (t, punt_TS),
push (0, despl) }
D { añade_ancho (cima (punt_TS), cima (despl)),
pop (punt_TS),
pop (despl) }

¿Cuándo puedo eliminar (borrar de memoria) una TS? Se puede eliminar cuando ya tenga hecha la traducción a código final del bloque que representa la tabla en cuestión. Tengo que mantenerla hasta entonces porque necesito conocer las direcciones relativas de las variables.

↳ En código máquina una variable se traduce como: dir_base + r,
↓
desplazamiento.

Normalmente sucede lo siguiente:



A la vez que genero el código intermedio voy generando el código objeto. Es por eso que puedo destruir la TS cuando llego al final de un bloque.

3. COMPROBACIÓN DE TIPOS.

En las reglas (acciones semánticas) de las expresiones hay que:

- Comprobar que los tipos de la expresión son correctos.
- Generar el código intermedio correspondiente.

En este apartado nos ocuparemos del primer punto, la comprobación de tipos: ¿Es correcta una expresión en cuanto a sus tipos?

* EXPRESIONES DE TIPO:

El tipo de una construcción de un lenguaje se denota (representa) mediante una expresión de tipos.

Expresiones de tipo:

1) **TIPO BÁSICO:** Un tipo básico es una expresión de tipo.

integer
real
Boolean
character
error_tipo → Tipo básico especial que indica un error durante la comprobación de tipos.
vacío ⟶ Tipo básico especial que indica ausencia de error durante la comprobación de tipos.
tipo_ok

2) **NOMBRE DE UN TIPO:** Como se pueden dar nombre a las expresiones de tipo, el nombre de un tipo es una expresión de tipo. Así, estos nombres pueden aparecer en expresiones de tipos donde antes sólo existían tipos básicos.

3) **TIPOS CONSTRUIDOS:** Un constructor de tipos aplicado a expresiones de tipo es una expresión de tipo.

Algunos tipos construidos son:

a) **Matrices:** Si T es una expresión de tipo, entonces:

array (I, T)

es una expresión de tipo que indica el tipo de una matriz con elementos de tipo T y conjunto de índices I.

Ejemplo: Var A: array [1..10] of integer ⇒ array (1..10, integer)

b) **Productos:** Si T_1 y T_2 son expresiones de tipo, entonces su producto cartesiano:

$T_1 \times T_2$

es también una expresión de tipo.

c) **Registros:** Se utiliza notación de producto cartesiano. La diferencia entre un registro y un producto es que los campos de un registro tienen nombres. Por tanto, el constructor de tipos "record" se aplicará a una tupla formada por nombres de campos y tipos de campos.

Ejemplo:

```
type fila = record
              direccion : integer;
              lexema : array [1..15] of Char;
end record;
```

⇓

record (dirección x integer) x (lexema x array (1..15, Char))

⇓

record (nombre_campo1 x tipo_campo1) x
(nombre_campo2 x tipo_campo2) x ... x
(nombre_campoN x tipo_campoN)

d) **Punteros:** Si T es una expresión de tipo, entonces:

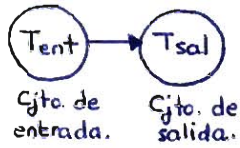
pointer (T)

es una expresión de tipo que indica el tipo "puntero a un objeto tipo T".

Ejemplo: Var P: ↑fila ⇒ pointer (fila)

e) **Funciones:** Dentro de los lenguajes de programación las funciones se pueden considerar como transformaciones de un dominio tipo D a un rango tipo R. La expresión de tipo:

$$D \to R \equiv (\text{tipo\_params\_entrada}) \to (\text{tipo\_salida})$$



indica el tipo de dicha función.

Cuando la función tiene más de un argumento, se utiliza el producto cartesiano.

Ejemplo: function f (a, b : char) return ↑ integer

⇓

(char x char) → pointer (integer)

ord_suma (x, y : integer) return char

⇓

(integer x integer) → char

function f(a, b : integer) return (c, d : integer)

⇓

(integer x integer) → (integer x integer)

Ejemplo: Var tabla : array [1..101] of fila

array (1..101 , record (dirección x integer) x (lexema x array (1..15, Char)))

Indice — Tipo

* SISTEMAS DE TIPO:

Un sistema de tipos es una serie de reglas para asignar expresiones de tipo a las distintas partes de un programa y verificar su corrección.

Las funciones principales del sistema de tipos son:

- Inferencia de tipos: Calcular y mantener información sobre los tipos de datos.
- Verificación de tipo: Asegurar la corrección de los tipos del programa atendiendo a las reglas definidas.

Ejemplo:

- "Si ambos operandos de los operadores aritméticos +, −, * son de tipo entero, entonces el resultado es de tipo entero".
- "El resultado del operador unario & es un puntero hacia el objetivo al que se refiere el operando. Si el tipo de operando es "..." el tipo de resultado es un puntero a "..."."

* COMPROBADOR DE TIPOS:

El comprobador de tipos implementa un sistema de tipos. Tiene como función asegurar que el tipo de una construcción coincida con el previsto en el contexto. Por ejemplo, que se sumen elementos del mismo tipo o tipos compatibles, no una matriz y una función.

Ejemplo:

E → num { E.tipo := entero }

E → true { E.tipo := boolean }

E → false { E.tipo := boolean }

E → id

```
{ id.tipo := busca_tipo ( id.posicion )
  if (id.tipo != NULL) then
     E.tipo := id.tipo
  else
     E.tipo := error_tipo }
```

Busca el id en la posición indicada de la TS.

/* Variable o su tipo no declarada */

Esto es correcto, pero debe escribirse de la siguiente manera:

```
{ id.tipo := busca_tipo ( id.posicion)
  E.tipo := if (id.tipo != NULL) then
                id.tipo
            else
                error_tipo
}
```

Ésta es la forma correcta porque tengo que poner:

atributo := forma de calcularlo

$E \rightarrow real$ { E.tipo := real }

$E \rightarrow \uparrow E_1$ { E.tipo := pointer (E1.tipo),
E.ancho := 2 }
↳ Ancho de palabra

DESPLAZAMIENTO
pointer => 2 (ancho de palabra).

$E \rightarrow Literal$ { E.tipo := character }

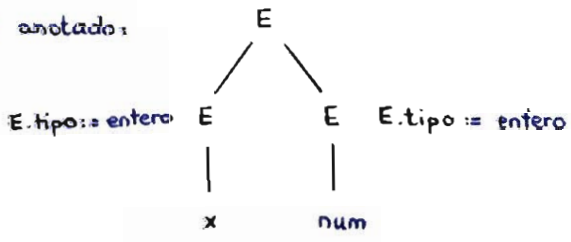
$E \rightarrow decimal$ { E.tipo := real }

$E \rightarrow E_1 \uparrow$ { E.tipo := if (E1.tipo == pointer(t)) then
t
else
error_tipo /* Error de tipo en puntero */
}

Árbol anotado :

$E \rightarrow E_1$ mod E_2 { E.tipo := if (E1.tipo == entero) and (E2.tipo == entero) then
entero
else
error_tipo /* Operando del tipo incorrecto */
}

Árbol anotado:

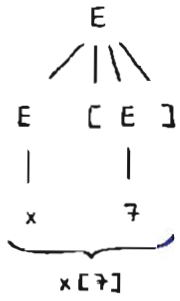


$E \rightarrow E_1 + E_2$ { E.tipo := if (E1.tipo == entero) and (E2.tipo == entero) then
entero
else
error_tipo
}

(También valdrían operandos reales y resultado real).

$E \rightarrow E_1 [E_2]$ { E.tipo := if (E_1.tipo == array(s,t)) and (E_2.tipo == entero) then
t
else
error_tipo /* Error de tipos en el array */
}

Árbol anotado:



$E \rightarrow E_1$ and E_2 { E.tipo := if (E_1.tipo == boolean) and (E_2.tipo == boolean) then
boolean
else
error_tipo
}

$E \rightarrow E_1$ op-rel E_2 { E.tipo := if (E_1.tipo == entero) and (E_2.tipo == entero) then
boolean
else
error_tipo
}

Si el lenguaje sólo permite operadores relacionales con enteros.

Si tuviera más de una tabla, para buscar una entrada en la TS tendría que pasarle el puntero de la tabla en la que debe buscar (en la actual, es decir, en la situada en la cima de la pila de punteros). Más adelante tendremos una función "Buscar" a la que le pasaremos un puntero a la tabla correspondiente y lo que quiero buscar (tipo, desplazamiento, ...)

Para las sentencias tengo el tipo tipo_ok. Las sentencias no las opero entre sí, simplemente me interesa saber si la sentencia es correcta.

Ejemplo:

Gramática:

P → D ; S ⇒ Declaraciones y sentencias.

1. S → id := E
2. S → if E then S_1
3. S → while E do S_1
4. S → S_1 ; S_2

Acciones semánticas para la comprobación de tipos en notación EDT (Esquema De Traducción):

```
1. S → id := E { S.tipo := if ( id.tipo == E.tipo) then
                              tipo_ok
                           else
                              error_tipo
               }
```

Está bien, pero hay que escribirlo con más detalle e indicar el acceso a la Tabla de Símbolos.

```
1. S → id := E { id.tipo := buscar ( id.posicion , TS , tipo)
                 S.tipo := if ( id.tipo != NULL) then
                              if ( id.tipo == E.tipo) then
                                 tipo_ok
                              else
                                 error_tipo     /* Incompatibilidad de tipos */
                           else
                              error_tipo        /* Variable o su tipo no declarada */
               }
```

```
2. S → if E then S1 { S.tipo := if (E.tipo == boolean) and (S1.tipo == tipo_ok) then
                                   tipo_ok
                                else
                                   error_tipo   /* Tipos incorrectos */
                    }
```

(1ª opción)

2. $S \rightarrow$ if E then S_1 { S.tipo := if (E.tipo == boolean) then
S_1.tipo
else
error_tipo /* Tipos incorrectos */
}

(2ª opción)

3. $S \rightarrow$ while E do S_1 { S.tipo := if (E.tipo == boolean) and (S_1.tipo == tipo_ok) then
tipo_ok
else
error_tipo /* Tipo incorrecto */
}

4. $S \rightarrow S_1 ; S_2$ { S.tipo := if (S_1.tipo == tipo_ok) and (S_2.tipo == tipo_ok) then
tipo_ok
else
error_tipo /* Tipos incorrectos */
}

El atributo "tipo" es un atributo sintetizado, voy pasando la información a través del árbol.

Si utilizásemos funciones en el lenguaje diseñado por la gramática, en la Tabla de Símbolos habría que añadir: los parámetros de la función, el tipo de estos parámetros, cómo se pasan estos argumentos y el tipo devuelto. Para los procedimientos tendríamos que hacer lo mismo.

* CONVERSIÓN DE TIPOS:

Como hemos visto, el comprobador de tipos tiene como misión verificar que la manipulación de tipos es correcta. La conversión de tipos depende del lenguaje y se da cuando hay dos operandos de distinto tipo. Por ejemplo, cuando tenemos dos enteros y una suma entre enteros, no hace falta ninguna conversión de tipos.

Hay 2 tipos de conversiones de tipos:

- Conversión implícita (coerción de tipo) → El compilador convierte automáticamente elementos de un tipo en elementos de otro tipo. La conversión se lleva a cabo en la acción semántica de la regla donde se realiza.

- Conversión explícita → El compilador no lo hace internamente, sino que es el programador quien indica el tipo destino. Funciona como una llamada a una función: recibe un tipo y devuelve otro.

En C, por ejemplo, al sumar un caracter y un entero se devuelve un valor de tipo numérico, pues el compilador de C transforma implícitamente el caracter en entero. En cambio, en Pascal esto hay que hacerlo explícitamente: el programador utilizará la función ord(c) para transformar el caracter en un número entero.

Ejemplo: Sin conversión automática de tipos (implícita).

```
E → E1 op_arit E2 { E.tipo := if (E1.tipo == E2.tipo) then
                                 E1.tipo
                             else
                                 error_tipo
                  }
```

↓

Si no hay conversión los elementos deben ser del mismo tipo o convertirse mediante funciones:

entero + entero } ok
real + real }

entero + real => intoreal(entero) + real
real + entero => real + intoreal(entero)

Conversión explícita.

Ejemplo: Con conversión automática de tipos (implícita).

Gramática:

1. E → num_ent { E.tipo := entero }

2. E → num_real { E.tipo := real }

3. E → id { id.tipo := busca_tipo (id.posicion)
   E.tipo := if (id.tipo != NULL) then
   id.tipo
   else
   error_tipo
   }

4. E → E_1 op_aritm E_2 { E.tipo := if (E_1.tipo == entero) and (E_2.tipo == entero) then
   entero
   else if (E_1.tipo == real) and (E_2.tipo == real) then
   real
   else if (E_1.tipo == real) and (E_2.tipo == entero) then
   real
   else
   error_tipo
   }

Otra forma más sencilla:

4. E → E_1 op_aritm E_2 { E.tipo := if (E_1.tipo == E_2.tipo) and (E_1.tipo != error_tipo) then
   E_1.tipo
   else
   real
   if (E_1.tipo == error_tipo) then
   error_tipo
   }

⚠ (De todas formas, es mejor la primera opción porque, si por ejemplo se añade E → literal a la gramática, la segunda forma ya no sería válida).

Ejemplo: Paso de parámetros por referencia.
Sin conversiones de tipo.
El analizador léxico (AL) no inserta los lexemas.

Gramática:

$P' \rightarrow P$

$P \rightarrow DP \mid FP \mid \lambda$

$D \rightarrow T\ id\ ;$

$F \rightarrow function\ id\ (L) : T\ ;\ begin\ B\ end$

$L \rightarrow T : id \mid L, L$

$T \rightarrow integer \mid real$

$B \rightarrow DB \mid SB \mid \lambda$

$S \rightarrow id := E\ ; \mid return\ E;$

$E \rightarrow id\ (A) \mid id$

$A \rightarrow E \mid A, A$

Acciones semánticas:

$P' \rightarrow \{ TSG := Crear\_TS();\ DesplG := 0 \}\ P$

$D \rightarrow T\ id\ ;\ \{$ if (TSL = NULL) then TS ← TSG ; Despl ← DesplG
else TS ← TSL ; Despl ← DesplL
if (Busca_TS (TS, id.lexema))
then Error /* Variable ya declarada */
else id.entrada := Inserta_TS (TS, id.lexema, T.tipo, Despl)
Despl := Despl + T.ancho $\}$

$F \rightarrow function\ id\ \{$ if (Busca_TS (TSG, id.lexema) ≠ NULL)
then Error /* Identificador ya declarado */
else id.entrada := Inserta_TS (TSG, id.lexema., A-proc)
TSL := Crear_TS(); Despl := 0 $\}$ (L) : T ; {
Inserta_Tipo TS (id.entrada, L.tipo → T.tipo) $\}$
begin B end { if (B.tipo_retorno ≠ T.tipo)
then Error /* Tipo de retorno incorrecto */
else Desactivar (TSL); TS := TSG $\}$

→

```
L → T : id { if (Busca_TS (TSL, id.lexema) ≠ NULL)
                then Error /* Parámetro ya declarado */
                else id.entrada := Inserta_TS (TSL, id.lexema, T.tipo, DesplL)
                     Despl := Despl + (2) → paso por referencia: ancho de palabra.
                     L.tipo := T.tipo }
```

```
L → L1 , L2 { L1.tipo := L1.tipo ⊗ L2.tipo }
```
→ para meter $p_1 \times p_2 \times p_3$ en el tipo de la función.

```
T → integer { T.tipo := entero , T.ancho := 4 }
```

```
E → id (A) { id.entrada := Busca_TS (TSG, id.lexema)
             if (id.entrada = NULL)
                then Error
                else if (Busca_Tipo TS (id.entrada) = A.tipo → t)
                        then E.tipo := t ;
                        else E.tipo := Error_tipo /* Tipos incorrectos en la llamada
                                                     a la funcion */
             }
```
→ valor devuelto por la función.

```
E → id { id.entrada := Busca_TS (id.lexema)
         if (id.entrada = NULL)
            then E.tipo := Error_tipo /* Variable no declarada */
            else E.tipo := Busca_Tipo TS (id.entrada)
```

```
A → A1 , A2 { A.tipo := A1.tipo * A2.tipo }
```

```
A → E { A.tipo := E.tipo }
```

Otras:

```
B → b B1 { B.TipoRet := B1.TipoRet }
B → λ { B.TipoRet := vacío }
```

```
S → return E { S.tipo := tipo_ok,
               S.TipoRet := E.tipo }
```

```
S → id := E { id.entrada := Busca_TS (id.lexema)
              if (id.entrada = NULL)
                 then S.tipo := error_tipo /* Identificador no declarado */
              else if (Buscar_Tipo TS (id.entrada) = E.tipo)
                      then S.tipo := tipo_ok;
                      else S.tipo := error_tipo /* Tipos incompatibles */ }
```

```
B → S B1 { if (S.Tipo_Ret = B1.Tipo_Ret)
              then B.Tipo_Ret := S.Tipo_Ret
           else if (B1.Tipo_Ret ≠ vacio)
                   then B.Tipo_Ret := B1.Tipo_Ret
                   else if (S.Tipo_Ret ≠ vacío)
                           then B.Tipo_Ret := S.TipoRet
                           else B.Tipo_Ret := Error }
                                /* Tipo erróneo
                                   en el return */
```

4. GENERACIÓN DE CÓDIGO INTERMEDIO.

Como ya hemos visto, en las reglas (acciones semánticas) de las expresiones hay que:

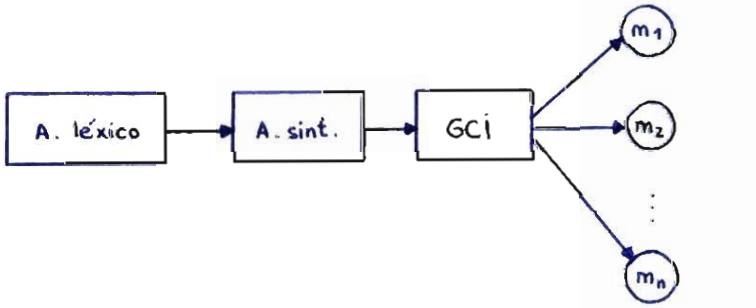
- Comprobar que los tipos de la expresión son correctos. (Apartado anterior)
- Generar el código intermedio correspondiente.

En este momento sabemos que el código es correcto. Ahora lo que hay que hacer es generar el código intermedio.

¿Por qué se utiliza código intermedio?

* Para reducir el salto entre la gramática del analizador sintáctico y el código máquina final.

* Para independizar el compilador de la máquina, de manera que el código final sólo hay que generarlo para la máquina concreta. El código intermedio es el mismo para todas las máquinas.



Código final

* Posibilidad de optimizar el código intermedio y obtener luego un código final optimizado.

Existen varios lenguajes (notaciones) intermedios:

1) Árbol sintáctico. (No suele caer en exámenes)
2) Notación polaca inversa (RPN).
3) Código de tres direcciones. (Importante)

Ejemplo: Notación polaca inversa (RPN)

a + b → a b +

a * b + c → a b * c +

if x then y := 7 → ¿Cómo es esto en RPN? La notación polaca inversa no es la notación más adecuada.

Si tengo código de 3 direcciones:

x := y op z → Dispongo de 3 direcciones: 2 para los operandos y 1 para el resultado.

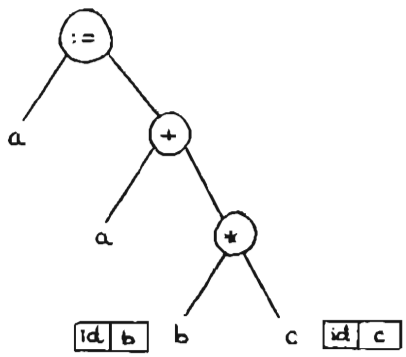
4.1. ÁRBOL SINTÁCTICO.

Esta representación gráfica se trata de un árbol tal que:

Nodos ⇒ Operadores

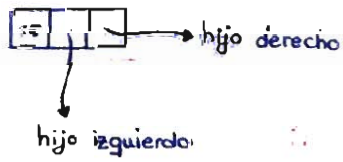
Hijos ⇒ Operandos

Ejemplo: a := a + b * c



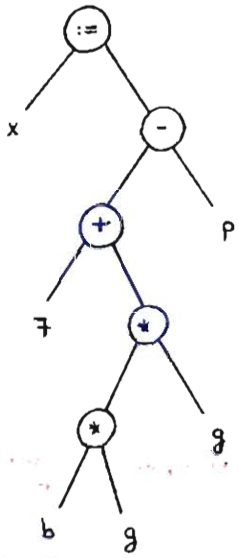
Este tipo de sentencias tienen una representación natural con árboles. Sin embargo, un bucle while o un bucle for no se representan así de manera tan sencilla.

En realidad, cada nodo es un registro:



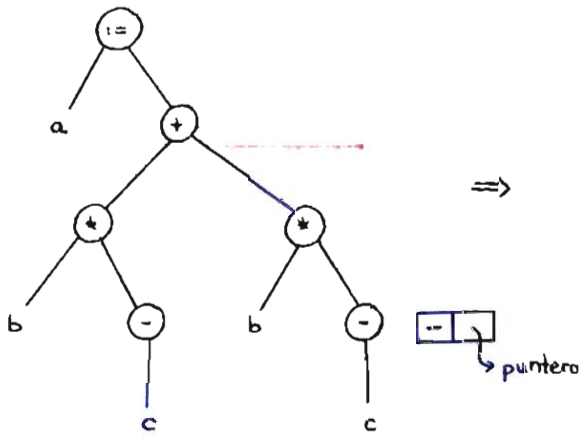
Los terminales son nodos hoja (registros también).

x := 7 + b * g * g - p

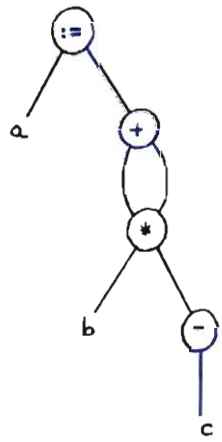


Los paréntesis no se incluyen en el árbol.

a := b * -c + b * -c



==>



Otra representación gráfica posible es el **árbol dirigido acíclico (GDA)**.

4.2. NOTACIÓN POLACA INVERSA (RPN).

Consiste en escribir primero los operandos y luego los operadores. Es una notación adaptada a lenguajes de programación funcional y buena para operadores aritméticos.

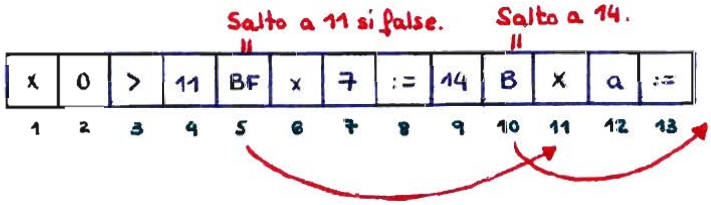
Ejemplo: a := a + b * c ⟶ b c * a + a :=

a * b + c → a b * c +

a + b * c → a b c * + → | b*c / a | Saca los 2 últimos operandos y opera.

if x>0 then x:=7 else x:=a → x 0 > x 7 := x a := if then else

La pega es que se miran todas las ramas, aunque sólo necesitaremos la del "if" o la del "else".



4.3. CÓDIGO DE 3 DIRECCIONES.

Este lenguaje intermedio utiliza sentencias tipo que tratan de representar todas las posibles sentencias del lenguaje. Se utiliza un máximo de 3 direcciones por instrucción.

Ejemplo:

Una sentencia tipo es, por ejemplo: x := y op z

La sentencia del lenguaje: a := a + b * c se transforma en:

$t_1 := b * c$
$a := a + t_1$

t_1 ≡ Variable temporal.

El compilador debe reservar espacio para las variables temporales.

Los tipos de proposiciones del código de 3 direcciones son los siguientes:

① x := y op a

Operaciones binarias: a + b
a * c
a - c

② x := op y

Sentencias unarias: Operadores unarios.
Negación lógica.

③ x := y

Copia directa de variables.

④ GOTO Etiqueta

Salto incondicional a Etiqueta (una sentencia concreta).

⑤ If x op-rel y GOTO Etiqueta

Salto condicional.

⑥ Param x
Call proc, n
Return y

Estas tres proposiciones forman una sentencia para las funciones:

* Param x => "x" es el parámetro de la función. Habrá una proposición de este tipo por cada parámetro que reciba la función.
* Call proc, n => Llamada a la función / procedimiento "proc", que recibe "n" parámetros.
* Return y => (Es opcional). Es lo que devuelve la función.

Ejemplo: función (X_1, X_2, X_3)

↓ Código de 3 direcciones.

Param X_1

Param X_2

Param X_3

Call función, 3

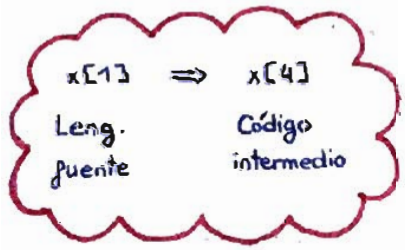
⑦ x := y[i]

x[i] := y

Indexación dentro de vectores. La "i" no son posiciones lógicas de la estructura, sino que suele ser un desplazamiento sobre la dirección base de la estructura.

Ejemplo:

Indice	0	1	2
Desplazam.	0	4	8
x	int	int	int



Integer ⇒ Desplazamiento = 2.

⑧ x := &y

x := *y

*x := y

El valor de "x" es igual a la dirección de "y", es decir, "x" es un puntero a "y".

Ejemplo: Código de 3 direcciones.

```
while  a > 0  do  a := a - 1
  ↓
comprobar:  if a < 0 GOTO salir
            a := a - 1
            GOTO comprobar
salir:  ...
```

5. CONEXIÓN A. SINTÁCTICO - GCI. REQUISITOS.

Ahora debemos conectar el analizador sintáctico con la generación de código intermedio. ¿Qué se necesita para ello?

① A todos los símbolos no terminales de la gramática se les añaden 2 atributos nuevos:

Sea $E \in N$ (símbolo no terminal). E tiene:

- E.lugar → Define una variable temporal.
- E.codigo → Almacena el código de 3 direcciones necesario para evaluar E.

② Función Nuevo_Temp(): Cada vez que se llama a esta función genera un nuevo temporal. En cada llamada utiliza un nombre distinto y reserva el espacio necesario.

③ Función Gen(_,_,_): Genera una sentencia de código de 3 direcciones en función de las direcciones y operandos que reciba como parámetros.

↓

Emite = Función que genera una nueva instrucción en un fichero temporal o en algún otro sitio.

Ejemplo: Código de 3 direcciones.

1. S → id := E
2. E → E + E
3. E → E * E
4. E → - E
5. E → (E)
6. E → id
7. S → while E do F

Gen ≈ Emite: DDS usa Gen
EDT usa Emite

Acciones necesarias para la generación del código intermedio:

1. S → id := E { S.codigo := E.código || (concatenar)
   Gen(busca_lugar_TS(id.entrada), ':=', E.lugar
   }

2. $E \rightarrow E_1 + E_2$ { E.lugar := Nuevo_Temp();
E.codigo := E_1.codigo || E_2.codigo ||
Gen (E.lugar, ':=', E_1.lugar, '+', E_2.lugar)
}

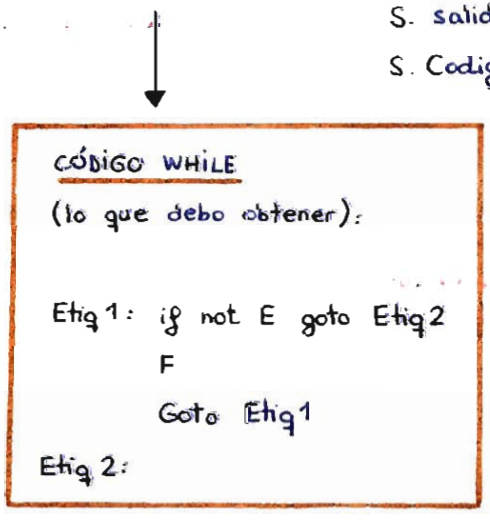
3. $E \rightarrow E_1 * E_2$ { E.lugar := Nuevo_Temp();
E.codigo := E_1.codigo || E_2.codigo ||
Gen (E.lugar, ':=', E_1.lugar, '*' E_2.lugar)
}

4. $E \rightarrow - E_1$ { E.lugar := Nuevo_Temp();
E.codigo := E_1.codigo ||
Gen (E.lugar, ':=', '-', E_1.lugar)
}

5. $E \rightarrow (E_1)$ { E.lugar := E_1.lugar;
E.codigo := E_1.codigo
}

6. E → id { E.codigo := ' ';
E.lugar := busca_lugar_TS (id_entrada)
}

7. S → while E do F { S.comienzo := Nueva_Etiqueta(); ≈ Etiq 1
S.salida := Nueva_Etiqueta(); ≈ Etiq 2
S.Codigo := Gen (S.comienzo, ':') || E.codigo ||
Gen ('if', E.lugar, '=', 'false', 'goto', S.salida) ||
F.codigo || Gen ('goto', S.comienzo) ||
Gen (S.salida, ':')

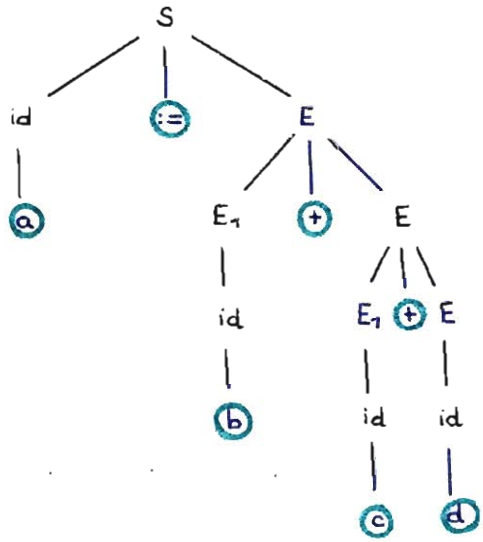


Ejemplo: Representación gráfica: Notación arborescente.

GCL: $S \rightarrow id := E$
$E \rightarrow E_1 + E$
$E_1 \rightarrow id$
$E \rightarrow id$

Sentencia a reconocer: a := b + c + d

Árbol sintáctico:



a := b + c + d

Ejemplo: Construir un árbol anotado DDS para generar código intermedio con salida un código de 3 direcciones.

1. $S \rightarrow id := E$

1. { id.lugar := buscar_lugar TS (id.entrada)
   S.codigo := E.codigo || Gen (id.lugar, ':=', E.lugar)

2. $E \rightarrow E_1 + E_2$

2. { E.lugar := Nuevo_Temp();
   E.codigo := E_1.codigo || E_2.codigo ||
   Gen (E.lugar, ':=', E_1.lugar, '+', E_2.lugar)

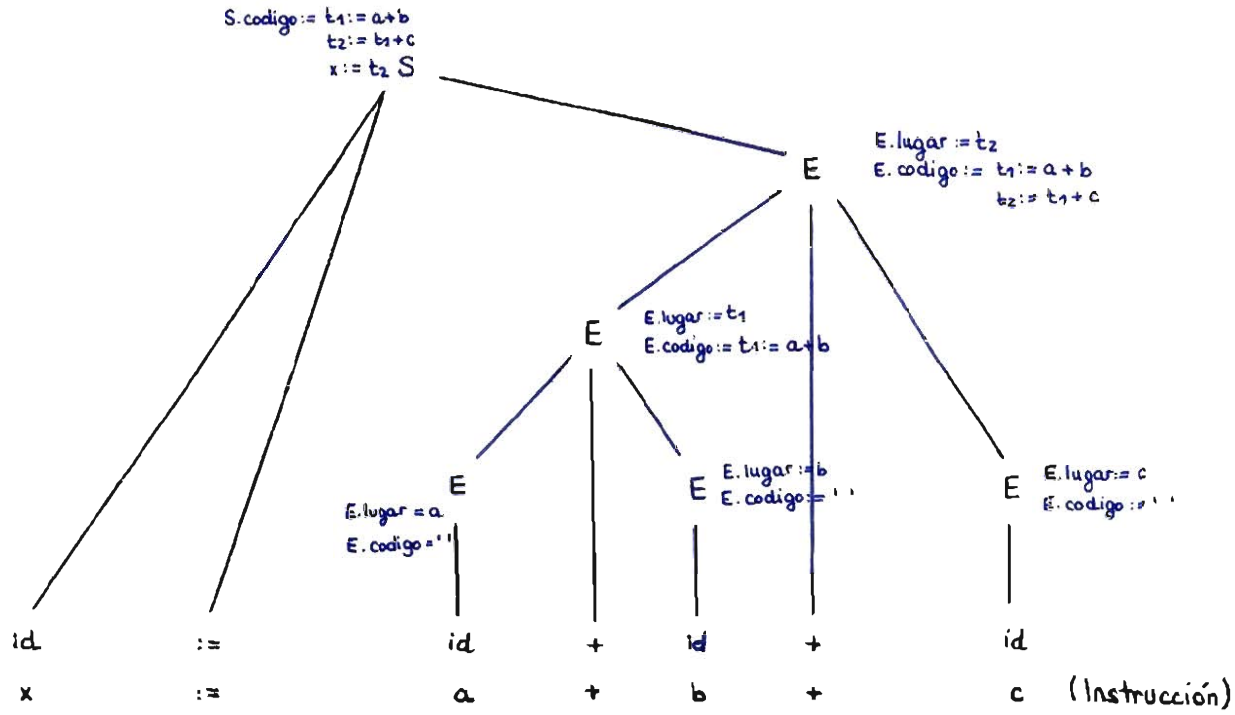
3. $E \rightarrow -E_1$

3. { E.lugar := Nuevo_Temp();
   E.codigo := E_1.codigo || Gen (E.lugar, ':=', '-', E_1.lugar)
   ↙ unario.

4. $E \rightarrow (E_1)$

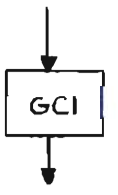
4. { E.lugar := E_1.lugar
   E.codigo := ' ' (vacío)

5. $E \rightarrow id$ 5. { E.lugar := id.lugar ≈ busca_lugar TS (id.entrada)
E.codigo := ' '



Cuando tengo el árbol anotado completo, en el nodo raíz S tengo la traducción completa de la entrada. Esta traducción es el valor del atributo S.codigo.

x := a + b + c



(Código de 3 direcciones)

$t_1 := a + b$

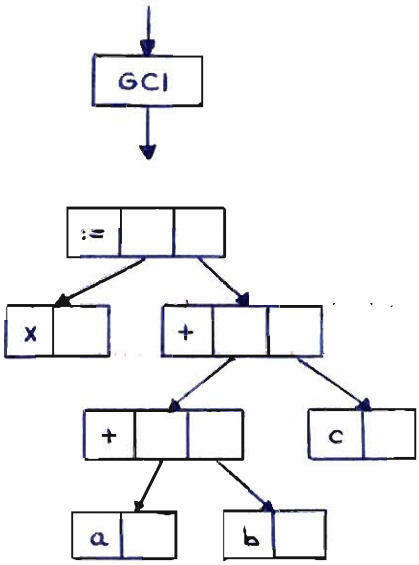
$t_2 := t_1 + c$

$x := t_2$

Ejemplo: Construir una DDS para generar código intermedio con salida un árbol sintáctico.

Lo que voy a obtener como salida del generador de código intermedio es lo siguiente: Nodos con punteros.

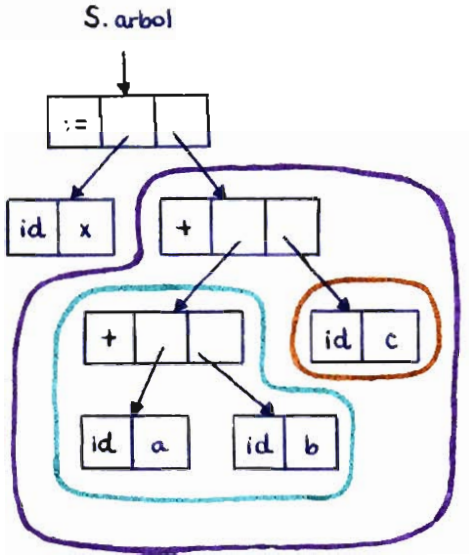
x := a + b + c



1. $S \rightarrow id := E$	1. id.lugar := buscar_lugar TS (id.entrada) S.arbol := crear_nodo (':=', crear_nodo_hoja ('id', id.lugar), E.arbol)
2. $E \rightarrow E_1 + E_2$	2. E.arbol := crear_nodo ('+', E_1.arbol, E_2.arbol)
3. $E \rightarrow - E_1$	3. E.arbol := crear_nodo_unario ('-', E_1.arbol)
4. $E \rightarrow (E_1)$	4. E.arbol := E_1.arbol
5. $E \rightarrow id$	5. id.lugar := buscar_lugar TS (id_entrada) E.arbol := crear_nodo_hoja ('id', id.lugar)

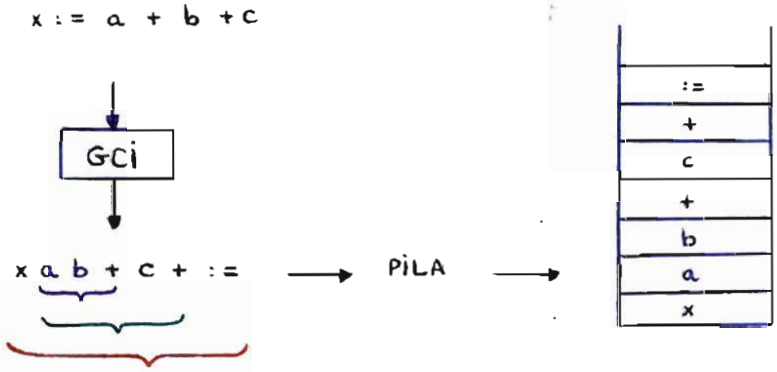
Al generar el código intermedio los paréntesis no sirven para nada. Se han usado para jerarquizar las operaciones en el lenguaje fuente, pero en el código intermedio no hay nada que jerarquizar, pues se realiza una operación cada vez.

Por tanto, tenemos :



En S.arbol tengo la traducción completa de la entrada en forma de árbol sintáctico.

Ejemplo: Construir una DDS para generar código intermedio con salida RPN (notación polaca inversa).

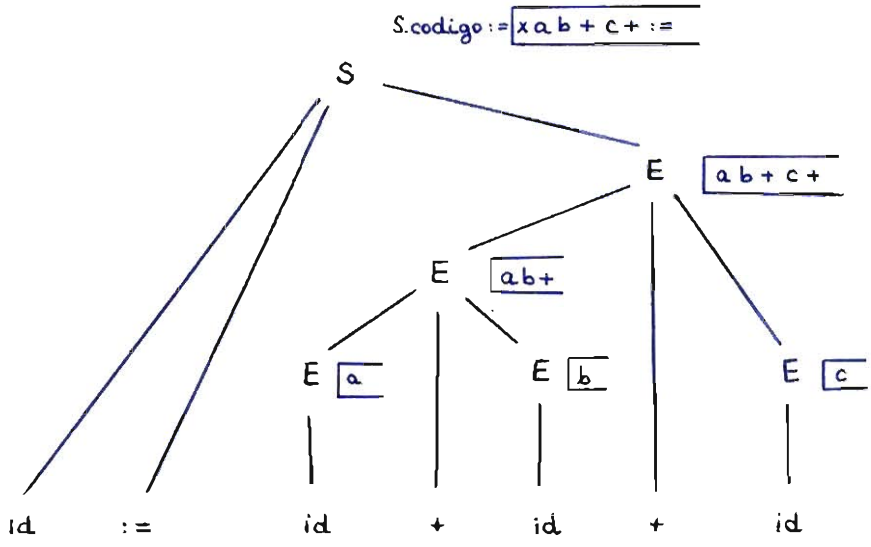


1. $S \rightarrow id := E$	1. id.lugar := buscar_lugar TS (id.entrada) S.codigo := push (id.lugar) ‖ E.codigo ‖ push (':=')
2. $E \rightarrow E_1 + E_2$	2. E.codigo := E_1.codigo ‖ E_2.codigo ‖ push ('+')
3. $E \rightarrow - E_1$	3. E.codigo := E_1.codigo ‖ push ('-')

4. $E \rightarrow (E_1)$ — 4. E.codigo := E_1.codigo

5. $E \rightarrow id$ — 5. id.lugar := buscar_lugar (id.entrada)
E.codigo := push (id.lugar)

Por tanto, tenemos:



En S.codigo tengo la traducción de la cadena de entrada en notación polaca inversa (RPN).

Ejemplo: Construir un EDT (esquema de traducción) para generar código de 3 direcciones.

1. S → id := E { id.lugar := buscar_lugar TS (id.entrada);
emite (id.lugar, ':=', E.lugar)
}

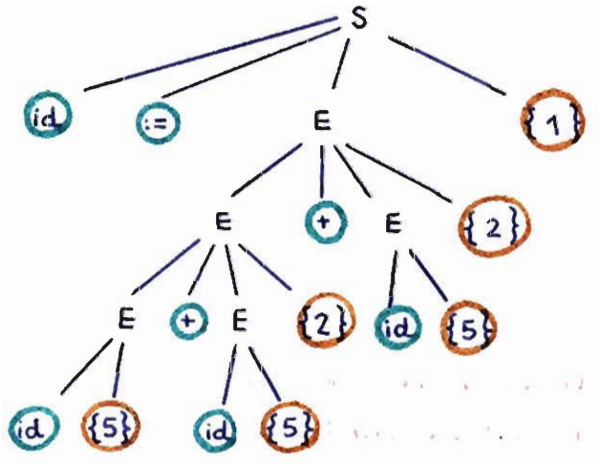
2. $E \rightarrow E_1 + E_2$ { E.lugar := Nuevo_Temp;
emite (E.lugar, ':=', E_1.lugar, '+', E_2.lugar)
}

3. $E \rightarrow -E_1$ { E.lugar := Nuevo_Temp();
   emite (E.lugar, ':=', '-', E_1.lugar)
   }

4. $E \rightarrow (E_1)$ { E.lugar := E_1.lugar }

5. E → id { id.lugar := buscar_lugar TS (id.entrada);
   E.lugar := id.lugar
   }

Árbol anotado EDT:



Fichero de salida:

$t_1 := a + b$

$t_2 := t_1 + c$

$x := t_2$

x := a + b + c

Como hemos visto, una Traducción Dirigida por Sintaxis consiste en que, dada una gramática de entrada y conocida la salida que queremos obtener (3 direcciones, RPN, árbol sintáctico), somos capaces de realizar la traducción.

5.1. SENTENCIAS Y EXPRESIONES.

Las **expresiones booleanas** pueden aparecer en el código de dos maneras:

* Computando un valor lógico (true, false).
* Como condición en una sentencia de control de flujo.

Igualmente hay dos posibles tratamientos de las expresiones booleanas:

* Representación numérica: Represento FALSO como 0 y TRUE como 1. También es válida cualquier otra representación numérica.

* Por control de flujo: Consiste en ejecutar una parte u otra de código en función de la expresión. Tan pronto como sepa qué instrucción tengo que ejecutar salto a la etiqueta correspondiente a dicha instrucción. Se almacena la dirección del THEN y el ELSE.

Se necesitan 2 atributos para almacenar las etiquetas:
E.verdad (dir. del THEN) y E.falso (dir del ELSE)

Ejemplo: Instrucción a < b

- Representación numérica:

```
if a < b then 1 else 0
```

- Representación por control de flujo:

```
if a < b then goto E.true else goto E.false
```

a < b: E = expresión que estoy evaluando

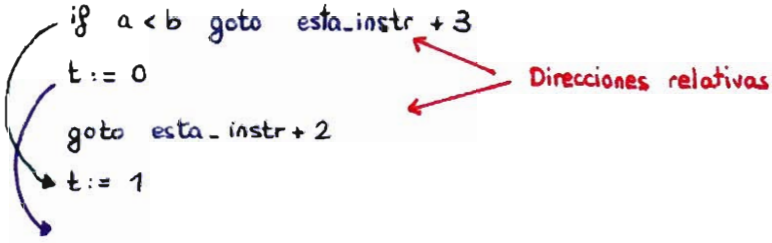
goto E.true: Salto a la instrucción etiquetada como E.true

goto E.false: Salto a la instrucción etiquetada como E.false

El valor que toma la expresión bodeana da lo mismo, lo importante es el salto.

Pasándolo a código de 3 direcciones:

- Representación numérica:



- Representación por control de flujo:

```
if a < b goto E.true
goto E.false
```

Ejemplo: Dada esta gramática, construir un EDT para generar código de 3 direcciones. Utilizar representación numérica para las expresiones booleanas.

1. $E \rightarrow E_1$ or E_2
2. $E \rightarrow E_1$ and E_2
3. $E \rightarrow$ not E_1
4. $E \rightarrow (E_1)$
5. $E \rightarrow id_1$ op_rel id_2
6. $E \rightarrow$ true
7. $E \rightarrow$ false

El EDT es:

1. $E \rightarrow E_1$ or E_2 { E.lugar := Nuevo_Temp();
   emite (E.lugar, ':=', E_1.lugar, 'or', E_2.lugar)
   }

2. $E \rightarrow E_1$ and E_2 { E.lugar := Nuevo_Temp();
   emite (E.lugar, ':=', E_1.lugar, 'and', E_2.lugar)
   }

3. $E \rightarrow not\ E_1$ { E.lugar := Nuevo_Temp();
emite (E.lugar, ':=', 'not', E_1.lugar)
}

4. $E \rightarrow (E_1)$ { E.lugar := E_1.lugar }

5. $E \rightarrow id_1$ op_rel id_2 { E := Nuevo_Temp();
emite ('if', id_1.lugar, op_rel, id_2.lugar, 'goto', esta_instr + 3);
emite (E.lugar, ':=', '0');
emite ('goto', esta_instr + 2);
emite (E.lugar, ':=', '1')
}

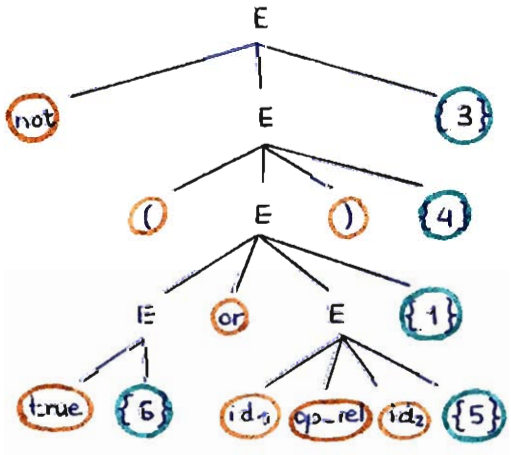
6. E → true { E.lugar := Nuevo_Temp();
emite (E.lugar, ':=', '1')
}

7. E → false { E.lugar := Nuevo_Temp();
emite (E.lugar, ':=', '0')
}

Para la expresión: not (true or a<b) se debería hacer lo siguiente:

* Meter 1 en un temporal (por el True)
* Hacer a < b y almacenarlo en un temporal.
* Hacer el or entre ambos temporales.
* Negar todo lo anterior.

Comprobamos que todo sea correcto:



Recorro el árbol de abajo a arriba y de izquierda a derecha.

En un fichero auxiliar se va generando el código de 3 direcciones:

6 { t_1 := 1
5 { if a < b goto esta_instr + 3
t_2 := 0
goto esta_instr + 2
t_2 := 1
1 { t_3 := t_1 or t_2
3 { t_4 := not t_3

Si en el código intermedio no existiera el operador AND y OR, podríamos hacer la traducción de la siguiente manera:

1. $E \rightarrow E_1$ or E_2 { E.lugar := Nuevo_Temp();
   emite (E.lugar, ':=', E_1.lugar, '+', E_2.lugar)
   }

→ Problema: 1 + 1 = 2

1. $E \rightarrow E_1$ or E_2 { E.lugar := Nuevo_Temp();
   emite ('if', E_1.lugar, '=', '1', 'goto', sig+3)
   emite ('if', E_2.lugar, '=', '1', 'goto', sig+2)
   emite (E.lugar, ':=', '0');
   emite ('goto', sig+1);
   emite (E.lugar, ':=', 1); }

2. $E \rightarrow E_1$ and E_2 { E.lugar := Nuevo_Temp();
   emite (E.lugar, ':=', E_1.lugar, '*', E_2.lugar)
   }

Ejemplo: Dada esta gramática, construir una DDS con control de flujo para expresiones booleanas. La salida debe ser código de 3 direcciones.

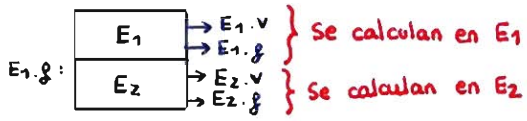
1. $E \rightarrow E_1$ or E_2
2. $E \rightarrow E_1$ and E_2
3. $E \rightarrow$ not E_1
4. $E \rightarrow (E_1)$
5. $E \rightarrow id_1$ op_rel id_2
6. $E \rightarrow$ true
7. $E \rightarrow$ false

Para resolver el problema nos vamos a apoyar en los diagramas de cajas.

→

Definición Dirigida por Sintaxis (DDS):

1. $E \rightarrow E_1$ or E_2



Las etiquetas E.verdad y E.falso son ATRIBUTOS HEREDADOS. (Son los primeros atributos heredados que aparecen).

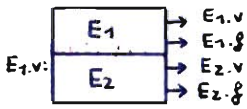
* Si E_1 es verdad, sé que es verdad $\Rightarrow$ Salto a E.verdad.
* Si E_1 es falso $\Rightarrow$ Salto a la primera instrucción de E_2.
* Si E_2 es verdad $\Rightarrow$ Salto a E.verdad.
* Si E_2 es falso $\Rightarrow$ Salto a E.falso.

1. $E \rightarrow E_1$ or E_2

1. $E_1.verdad := E.verdad$
   $E_1.falso := Nueva\_Etiqueta();$
   $E_2.verdad := E.verdad$
   $E_2.falso := E.falso$
   $E.codigo := E_1.codigo \,||\, Gen(E_1.falso, ':') \,||\, E_2.codigo$

Vamos concatenando el código.

2. $E \rightarrow E_1$ and E_2



* Si E_1 es verdad $\Rightarrow$ Salto a la primera instrucción de E_2.
* Si E_1 es falso, sé que es falso $\Rightarrow$ Salto a E.falso.
* Si E_2 es verdad $\Rightarrow$ Salto a E_2.verdad.
* Si E_2 es falso $\Rightarrow$ Salto a E.falso.

2. $E \rightarrow E_1$ and E_2

2. $E_1.falso := E.falso$
   $E_1.verdad := Nueva\_Etiqueta();$
   $E_2.verdad := E.verdad$
   $E_2.falso := E.falso$
   $E.codigo := E_1.codigo \,||\, Gen(E_1.verdad, ':') \,||\, E_2.codigo$

→

③ $E \rightarrow not\ E_1$

* Si E_1 es verdad ⇒ Salto a E.falso.
* Si E_2 es falso ⇒ Salto a E.verdad.

3. $E \rightarrow not\ E_1$

3. $E_1.verdad := E.falso$
   $E_1.falso := E.verdad$
   $E.codigo := E_1.codigo$

④ $E \rightarrow (E_1)$

No sucede nada. Los paréntesis no sirven de nada al generador de código intermedio.

4. $E \rightarrow (E_1)$

4. $E_1.verdad := E.verdad$
   $E_1.falso := E.falso$
   $E.codigo := E_1.codigo$

⑤ $E \rightarrow id_1\ op\_rel\ id_2$

5. E.codigo := Gen ('if', busca_lugar TS (id_1.entrada), op_rel, busca_lugar TS (id_2.entrada), 'goto', E.verdad) ; ||
   Gen ('goto', E.falso) ;

(id_1.lugar, id_2.lugar)

Ejemplo: a > b se traduce como: if a > b goto E.verdad
goto E.falso

⑥ $E \rightarrow true$

6. E.codigo := Gen ('goto', E.verdad)

⑦ $E \rightarrow false$

7. E.codigo := Gen ('goto', E.falso)

(8.) Si en una gramática sólo con booleanos (id = true o false) tuviéramos la regla sintáctica:

$E \rightarrow id$

La DDS sería:

E.codigo := Gen ('if', id.lugar, '=', '1', 'goto', E.verdad) ||
Gen ('goto', E.falso)

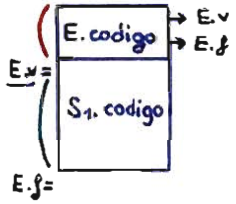
Si la gramática tuviera, por ejemplo, enteros y booleanos habría que diferenciarlos en la DDS:

```
id.tipo := buscar_tipo ( id.entrada) ;
if (id.tipo = entero) then
    E.lugar := id.lugar
    E.tipo := id.tipo
    E.codigo := ' '
else if (id.tipo = boolean) then
    Gen ('if', id.lugar, 'goto', E.verdad)
    Gen ('goto', E.falso)
else
    E.tipo := tipo_error;
```

Ejemplo: Realizar una DDS con control de flujo para expresiones booleanas y cuya salida sea un código de 3 direcciones.

1. $S \rightarrow$ if E then S_1
2. $S \rightarrow$ if E then S_1 else S_2
3. $S \rightarrow$ while E do S_1
4. $E \rightarrow$ id
5. $S \rightarrow$ id := E

1. S → if E then S_1

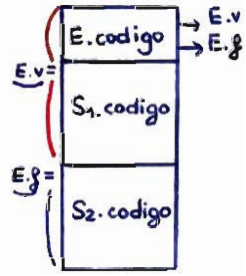


* Si E se cumple => Salto al comienzo de S_1.
* Si E no se cumple => Salto a la instrucción siguiente de S. Para ello utilizo el atributo S.siguiente.

1. S → if E then S_1

1. E.verdad := Nueva_Etiqueta();
   E.falso := S.siguiente
   S.codigo := E.codigo || Gen(E.verdad, ':') || S_1.codigo
   S_1.siguiente := S.siguiente

2. S → if E then S_1 else S_2

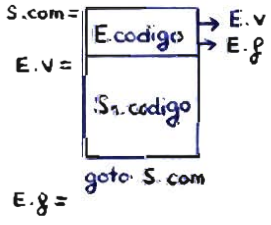


* Si E se cumple => Salto al comienzo de S_1.
* Si E no se cumple => Salto al comienzo de S_2.

2. S → if E then S_1 else S_2

2. E.verdad := Nueva_Etiqueta();
   E.falso := Nueva_Etiqueta();
   S.codigo := E.codigo || Gen(E.verdad, ':') || S_1.codigo ||
   Gen('goto', S.siguiente) ||
   Gen(E.falso, ':') || S_2.codigo

3. S → while E do S_1



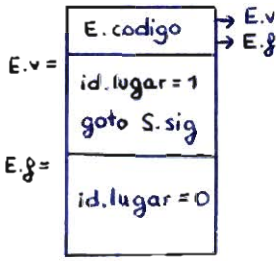
3. E.verdad := Nueva_Etiqueta();
   E.falso := S.siguiente
   S.comienzo := Nueva_Etiqueta();
   S_1.siguiente := S.comienzo // Para el bucle while.
   S.codigo := Gen(S.comienzo, ':') || E.codigo ||
   Gen(E.verdad, ':') || S_1.codigo ||
   Gen("goto", S.comienzo)

→

(4.) E → id

4. E.codigo := Gen ('if', busca_lugarTS(id.entrada), '=', '1', 'goto', E.verdad) ||
Gen ('goto', E.falso)

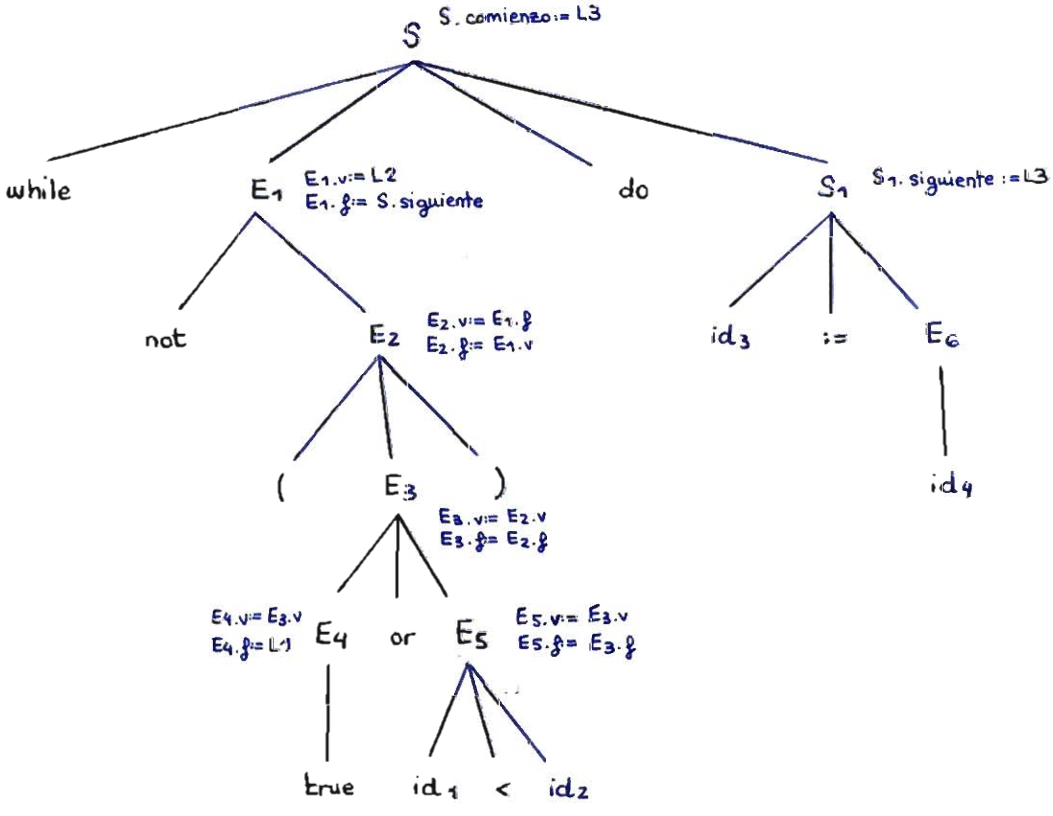
(5.) S → id := E



5. id.lugar := buscar_lugarTS (id.entrada)
E.verdad := Nueva_Etiqueta ();
E.falso := Nueva_Etiqueta ();
S.codigo := E.codigo || Gen (E.verdad, ':') ||
Gen (id.lugar, ':=', '1') ||
Gen ('goto', S.siguiente) || Gen (E.falso, ':') ||
Gen (id.lugar, ':=', '0')

Aplicación: while not (true or a < b) do
a := b

Hacemos el árbol anotado DDS :



$E_4.codigo := goto\ E_4.verdad$ → $E_3.verdad$

$E_5.codigo := if\ a < b\ goto\ E_5.verdad$ → $E_3.verdad$
$goto\ E_5.falso$

$E_2.codigo := E_3.codigo := goto\ E_3.verdad$
$L1 : if\ a < b\ goto\ E_3.verdad$
$goto\ E_3.falso$

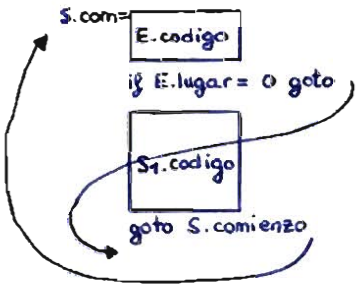
$E_1.codigo := E_2.codigo$

$S_1.codigo := a := b$

$S.codigo :=$ L3 : goto S.siguiente
L1 : if a < b goto S.siguiente } $E_1.codigo$
goto L2
L2 : a := b } $S_1.código$
goto L3 } Salto al comienzo

Ejemplo : Representación numérica (RN) para las expresiones booleanas vs Control de flujo (CF)
Codigo intermedio : Código de 3 direcciones.

① S → while E do S_1



DDS RN :

1. S.comienzo := Nueva_Etiqueta ();
   $S_1.siguiente := S.comienzo$
   S.codigo := Gen (S.comienzo, ':') || E.codigo ||
   Gen ('if', E.lugar, '=', '0', 'goto',
   S.siguiente) || $S_1.codigo$ ||
   Gen ('goto', S.comienzo)

EDT RN :

1. S → while { S.comienzo := Nueva_Etiqueta ();
   emite (S.comienzo, ':') } E do
   { $S_1.siguiente := S.comienzo$,
   emite ('if', E.lugar, '=', '0', 'goto',
   S.siguiente) } S_1 { emite ('goto', S.comienzo) }

DDS CF: (Visto también en la página 97)

1. S.comienzo := Nueva_Etiqueta ();
   E.verdad := Nueva_Etiqueta ();
   E.falso := S.siguiente
   S_1.siguiente := S.comienzo // Para el bucle while.
   S.codigo := Gen (S.comienzo, ':') || E.codigo || Gen (E.verdad, ':') ||
   S_1.codigo || Gen ('goto', S.comienzo)

EDT CF:

1. S → while { S.comienzo := Nueva_Etiqueta (); emite (S.comienzo, ':');
   E.verdad := Nueva_Etiqueta (); E.falso := S.siguiente } E do
   { emite (E.verdad, ':'), S_1.siguiente := S.comienzo } S_1
   { emite ('goto', S.comienzo) }

② S → for id = 1 to n do S_1

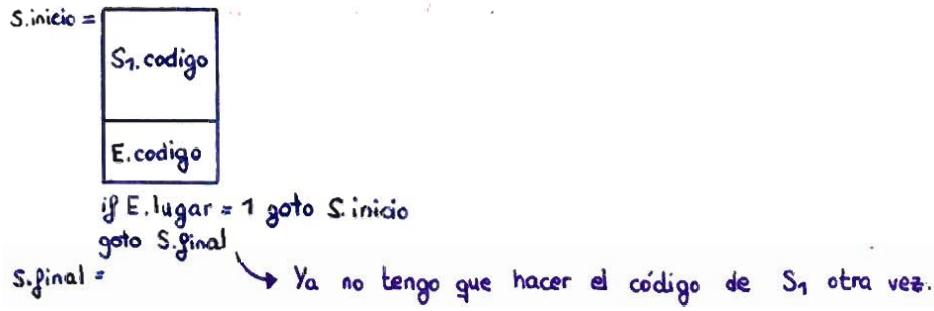
Código a obtener →
```
id := 0
S.inicio : id++
if id > n goto S.siguiente
S1
goto S.inicio
```

EDT RN:

2. S → for id = 1 to n do { S.inicio := Nueva_Etiqueta ();
   emite (id.lugar, ':=', '0'), emite (S.inicio, ':'),
   emite (id.lugar, ':=', id.lugar, '+', '1'),
   emite ('if', id.lugar, '>', n.valor, 'goto', S.siguiente),
   S_1.siguiente := S.inicio } S_1 { emite ('goto', S.inicio) }

Ejemplo: Representación numérica (RN) vs. control de flujo (CF).
Construir la DDS para la siguiente expresión booleana:

1. $S \rightarrow$ repeat S_1 until E (RN)



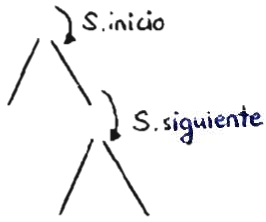
1. $S \rightarrow$ repeat S_1 until E

1. S.inicio := Nueva_Etiqueta ();
   S.final := Nueva_Etiqueta ();
   S.codigo := Gen (S.inicio, ':') || S_1.codigo ||
   E.codigo || Gen ('if', E.lugar, '=', '1',
   'goto', S.inicio) || Gen ('goto', S.final) ||
   Gen (S.final, ':')

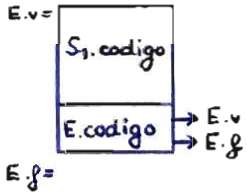
Las etiquetas S.inicio y S.final son LOCALES a la regla. Es la forma más sencilla de hacerlo.

Otra forma elegante de hacer la etiqueta S.final es utilizar el atributo heredado S.siguiente.

A la etiqueta S.siguiente se le da valor al inicio del programa e irá pasando de padres a hijos como atributo heredado.



1. $S \rightarrow$ repeat S_1 until E (CF)



1. E.verdad := Nueva_Etiqueta ();
   E.falso := S.siguiente ;
   S_1.siguiente := E.verdad ; // Para el bucle repeat.
   S.codigo := Gen (E.verdad, ':') || S_1.codigo ||
   E.codigo

6. UNIÓN A. SINTÁCTICO - TDS.

Ahora vamos a ver cómo podemos juntar el análisis sintáctico con la Traducción Dirigida por Sintaxis (TDS).

6.1. ANALIZADOR SINTÁCTICO DESCENDENTE RECURSIVO PREDICTIVO CON UN ESQUEMA DE TRADUCCIÓN (EDT) (Los atributos pueden ser heredados y sintetizados)

Algoritmo:

↳ heredados por la izquierda.

Entrada:

Gramática LL(1)
Esquema De Traducción (EDT) para dicha gramática

Salida:

Código del analizador sintáctico + generador de código intermedio

Procedimiento:

a) Para cada símbolo no terminal A se construye una función que tenga un parámetro de entrada por cada atributo heredado de A y que como salida devuelva los valores de los atributos sintetizados de A.

b) El código de cada función dependerá de la regla y de los elementos de la parte derecha de la regla:

1) Encontramos un token X → El token X tiene un atributo X.x. Se guarda el atributo X.x en una variable local de la función. Si el token concuerda con el símbolo de pre-análisis entonces se llama al analizador léxico para el siguiente token. ↳ token en la entrada.

2) Encontramos un símbolo no terminal B → Se genera una instrucción del tipo $s := B(b_1, b_2, \ldots, b_n)$ donde los b_i son cada uno de los atributos heredados de B y s es una variable local donde almacenar los atributos sintetizados que devuelva B.

3) Encontramos una acción semántica → Se implementa la acción semántica dentro de la función, reemplazando las referencias a los atributos por las variables correspondientes.

→

Ejemplo:

DT: $D \rightarrow T \{ L.tipo := T.tipo \} L ;$

$T \rightarrow int \{ T.tipo := entero \}$

$T \rightarrow float \{ T.tipo := float \}$

$L \rightarrow id \{ añade\_tipoTS(id.entrada, L.tipo, R.tipo := L.tipo \} R$

$R \rightarrow ; \{ L.tipo := R.tipo \} L$

$R \rightarrow \lambda \{\}$

Símbolos no terminales: D, T, L, R

Atributos:

D → No tiene atributos.

T → Heredados = ϕ ; Sintetizados = T.tipo.

L → Heredados = L.tipo ; Sintetizados = ϕ.

R → Heredados = R.tipo ; Sintetizados = ϕ.

```
FUNCTION T () : tipo ──→ T no tiene atributos heredados ⇒ No tiene parámetros.
{                        T devuelve 'tipo' (atributo sintetizado)
    var t : tipo;                                          } Token.
    if (siguiente_token == 'int') then                     }
    {                                                      }
        equipara ('int');                                  }
        t := entero;                                       }
    }                                                      }
    else if (siguiente_token == 'float') then              } Acción semántica + comprobar pre-análisis.
    {                                                      }
        equipara ('float');                                }
        t := real;                                         }
    }                                                      }
    else                                                   }
    {                                                      }
        error;                                             }
    }                                                      }
    return t;                                              }
}                                                          }
```

→

```
PROCEDURE L ( t : tipo)
{
    if ( siguiente_token == 'id') then
    {
        añade_tipoTS ( siguiente_token.entrada, t);
        equipara ('id');
        R(t);
    }
    else
    {
        error;
    }
}
```

L no devuelve nada porque L no tiene atributos sintetizados. Por eso es un procedure y no una función.

```
PROCEDURE R ( t : tipo)
{
    if (siguiente_token == ';') then
    {
        equipara (';');
        L(t);
    }
}

PROCEDURE D ()
{
    var t : tipo ;
    t := T();
    L(t);
    equipara (';')
}

                    // empareja, valida
PROCEDURE Equipara ( t : token)
BEGIN
    if ( siguiente_token = t) then
        siguiente_token = scan ();      // Leer siguiente token.
    else
        error (t);
    end if;
END
```

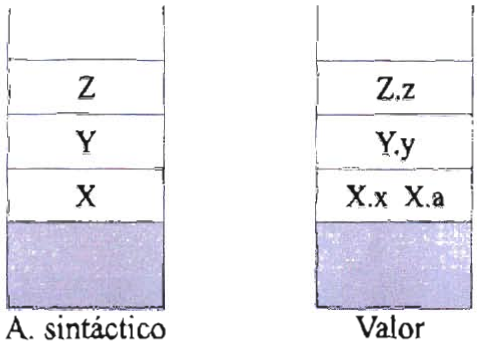
6.2. ANALIZADOR SINTÁCTICO ASCENDENTE LR CON ATRIBUTOS SINTETIZADOS

↳ sólo sintetizados.

Voy a utilizar 2 pilas paralelas:

- Pila del analizador sintáctico: Contiene símbolos no terminales y estados. ↗ símbolos terminales
- Pila valor: Contiene los valores de los atributos sintetizados de los símbolos que están en la pila del analizador sintáctico.

↳ o pila Atributos.

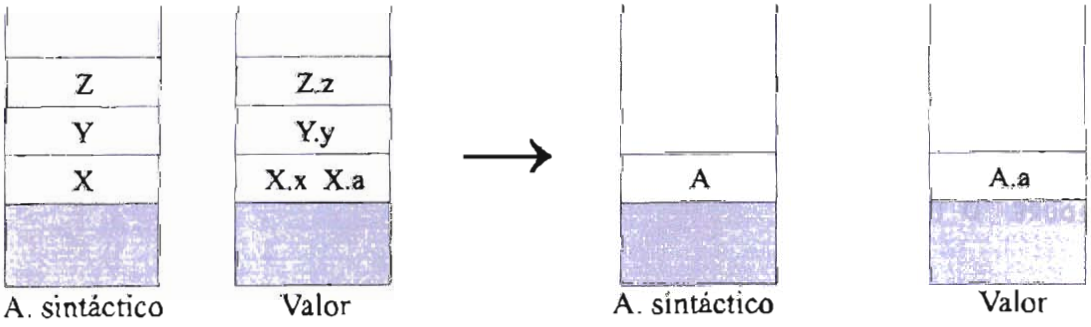


Por ejemplo, tengo la regla:

$A \rightarrow XYZ$ $\quad A.a := f(X.x, Y.y, Z.z)$ → A.a depende sólo de atributos de los hijos.

⇓ Atributo sintetizado.

Cuando determino que XYZ es el pivote (subcadena que se equipara con la parte derecha de una regla para hacer una reducción), ejecuto la acción para calcular A.a antes de quitar de la pila los datos necesarios para ello. Después, almaceno A y su atributo A.a en las pila correspondiente.



Ejemplo: Tengo la siguiente gramática con sus reglas semánticas.

1. $S \rightarrow id := E$ $\quad$ 1. S.tipo := if (buscar_tipoTS (id.entrada) == E.tipo) then
   vacío = tipo_ok
   else
   tipo_error

2. $E \rightarrow E_1 > T$ $\quad$ 2. E.tipo := if (E.tipo == T.tipo ∈ {int, real}) then
   booleano
   else
   tipo_error

3. $E \rightarrow T$ 3. E.tipo := T.tipo

4. $T \rightarrow (E)$ 4. T.tipo := E.tipo

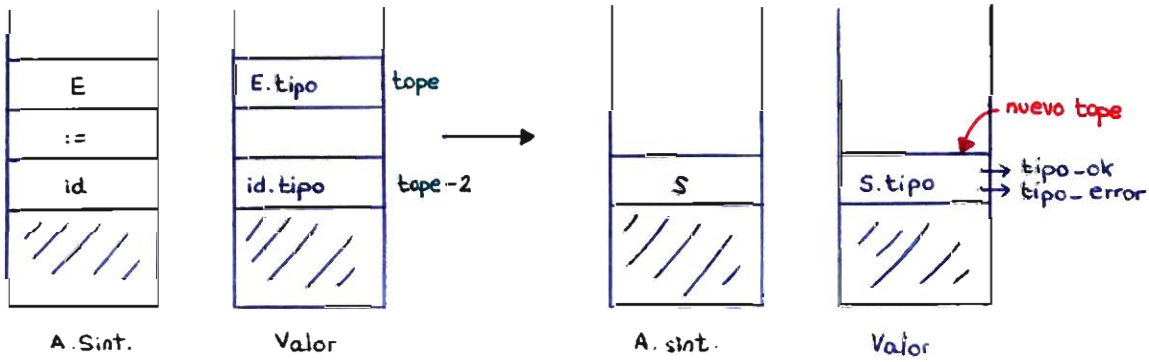
5. $T \rightarrow id$ 5. T.tipo := buscar_tipo TS (id.entrada)

Ahora vamos a hacer el programa, los fragmentos de código de cada regla:

1. val [ntope] := if (buscar_tipoTS (val [tope - 2]) == val [tope]) then
   vacío ≡ tipo_ok
   else
   tipo_error

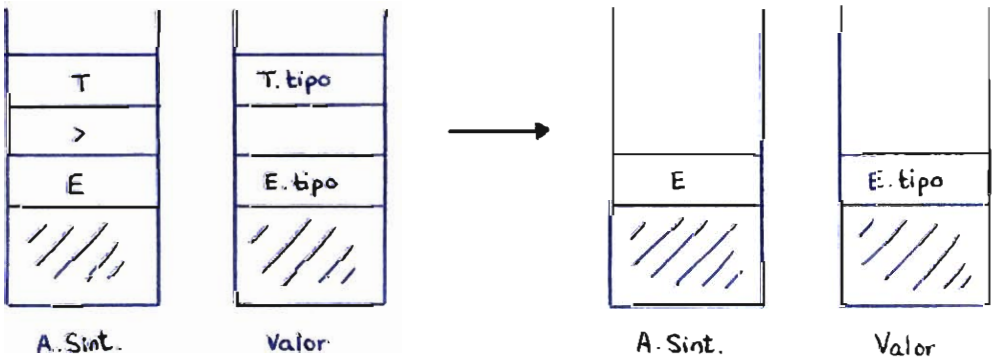
Nuevo tope de la pila de atributos

Situación de las pilas:



2. val [ntope] := if (val [tope - 2] == val [tope] ∈ { int , real }) then
   booleano
   else
   tipo_error

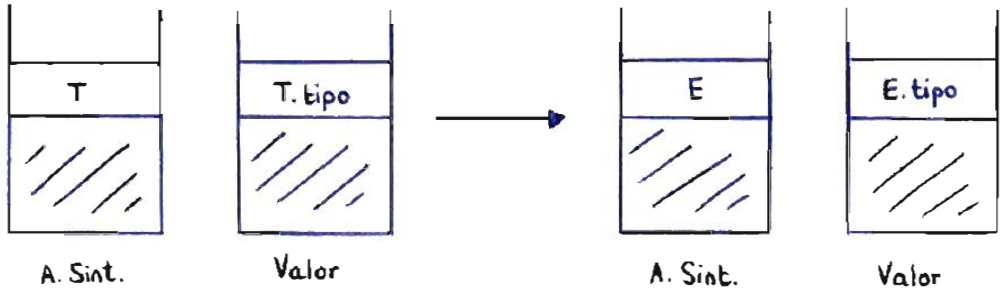
Situación de las pilas:



3. val [ntope] := val [tope]

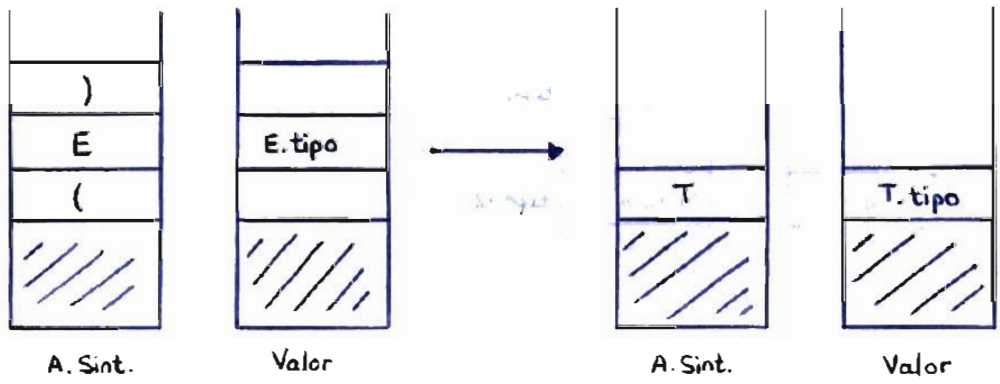
Este código no es necesario porque el valor es el mismo. Por tanto, lo puedo dejar en blanco.

Situación de las pilas:



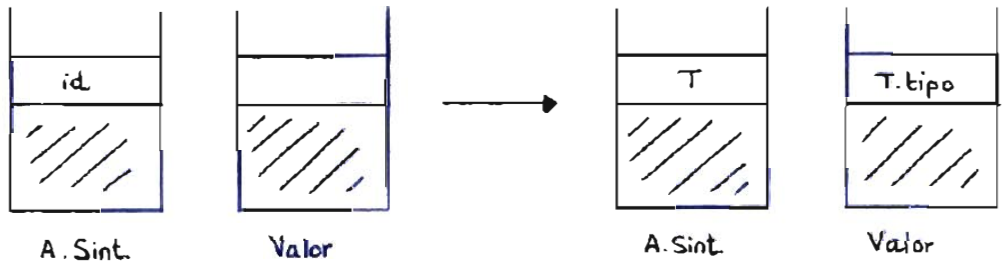
4. val [ntope] := val [tope - 1]

Situación de la pila:



5. val [ntope] := buscar_tipo TS (val [tope])

Situación de la pila:



6.3. ANALIZADOR SINTÁCTICO ASCENDENTE LR CON ATRIBUTOS SINTETIZADOS Y HEREDADOS POR LA IZQUIERDA

- Atributos sintetizados.

Se tratan de la misma manera que en el método anterior (Analizador sintáctico ascendente LR sólo con atributos sintetizados).

- Atributos heredados.

Los atributos heredados que aceptan los algoritmos ascendentes son limitados. En LR el padre (antecedente de la regla) no se mete en la pila hasta que se han reconocido todos sus hijos. Por tanto, no es posible que un hijo herede atributos de su padre. Sólo se podrá heredar de símbolos de la parte derecha de la regla (consecuente) que estén actualmente en pila, es decir, se puede heredar de los hermanos izquierdos del símbolo actual. → **Sólo se admiten atributos heredados por la izquierda.**

Importante: El valor del atributo heredado de un símbolo X se calcula en una acción justo antes del símbolo X.

Se pueden dar diferentes casos de atributos heredados:

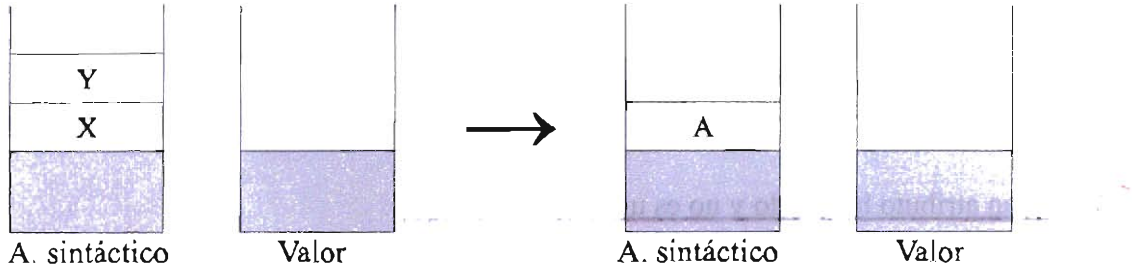
1. El atributo heredado se calcula como una copia del sintetizado.

Se tiene una regla del tipo:

$$A \rightarrow XY \qquad Y.h := X.s$$

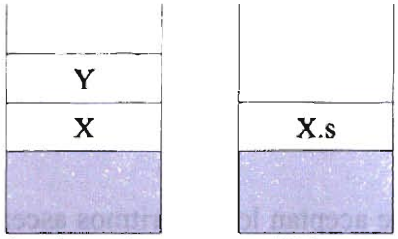
donde Y.h es un atributo heredado del atributo sintetizado X.s. X es hermano por la izquierda de Y.

La situación de la pila es la siguiente:



¿Dónde, en qué posición de la pila Valor, coloco el valor del atributo heredado Y.h?
No hay espacio en la pila para Y.h.

Si el atributo heredado se calcula como una copia de un atributo sintetizado, siempre que aparezca el atributo heredado se sustituirá por el sintetizado, es decir, siempre que quiera acceder al valor del heredado accederé en su lugar al valor del sintetizado, pues de éste sí se conoce la posición de la pila. Así, del atributo heredado nos olvidamos y eliminamos las asignaciones entre el heredado y el sintetizado, pues ya lo hemos sustituido en las acciones.



2. Imposibilidad de predecir la posición de la pila en la que se encuentra el atributo sintetizado del que depende el atributo heredado.

Se tiene la gramática:

1. S → aAC	C.h := A.s	} Esto es el problema
2. S → bABC	C.h := A.s	
3. C → c	C.s := f(C.h)	

Ahora el problema es que no puedo predecir en qué posición de la pila de valores se encuentra el atributo sintetizado.

En la gramática anterior, el valor del atributo sintetizado (A.s) puede aparecer en la posición [tope-1] (regla 1) o en la posición [tope-2] (regla 2).

Solución: Introducir un marcador que permitirá conocer siempre la posición del atributo sintetizado. M es el marcador y siempre deriva en λ.

Cambio la gramática de la siguiente manera:

1. S → aAC	C.h := A.s
2. S → bABMC	M.h := A.s, C.h := M.s
3. C → c	C.s := f(C.h)
4. M → λ	M.s := M.h

3. Hay un atributo heredado y no es una regla de copia.

Se tiene una regla del tipo:

S → aAC C.h := f(A.s)

Solución: Introducir un marcador que copie el valor del atributo sintetizado. Será en la acción semántica del marcador donde se calcule la función.

Cambio la gramática de la siguiente manera:

1. S → aAMC	M.h := A.s, C.h := M.s
2. M → λ	M.s := f(M.h)

↳ Aquí es donde hago el trabajo.

Ejemplo CASO 1: Atributos heredados.

Gramática:	DDS:	
1. D → TL ;	1. L.tipo := T.tipo	Heredados
2. T → int	2. T.tipo := entero	
3. T → float	3. T.tipo := real	Sintetizados
4. L → L_1 , id	4. L_1.tipo := L.tipo añadir_tipoTS (id.entrada , L.tipo)	
5. L → id	5. añadir_tipoTS (id.entrada , L.tipo)	

El atributo L.tipo es heredado (depende de su hermano izquierdo T) y se calcula como una copia del atributo sintetizado T.tipo. Por tanto, se elimina la asignación de copia y se sustituye L.tipo por T.tipo en toda la DDS.

Por la misma razón se elimina también la asignación L_1.tipo := L.tipo

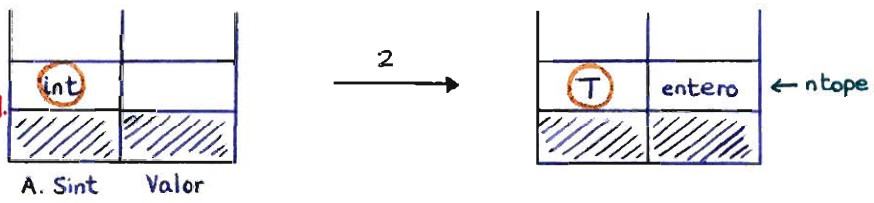
Las acciones quedan de la siguiente manera:

1. // No se hace nada.
2. T.tipo := entero
3. T.tipo := real
4. añadir_tipoTS (id.entrada , T.tipo)
5. añadir_tipoTS (id.entrada , T.tipo)

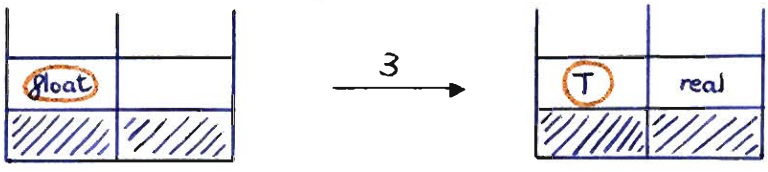
Código de la implementación:

2. val [ntope] = entero

Pila valor.
También podría decirse atrib[ntope].

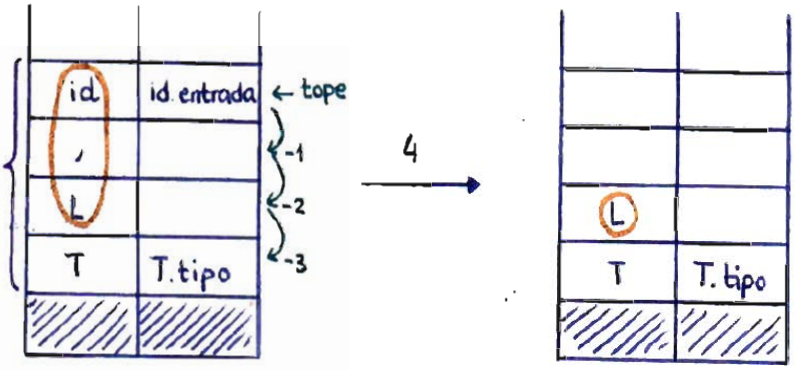


3. val [ntope] = real

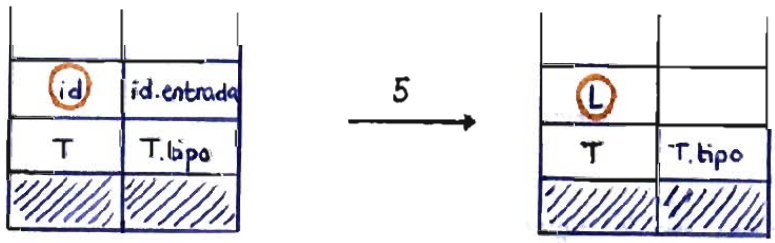


4. añadir_tipo TS (val[tope] , val [tope - 3])

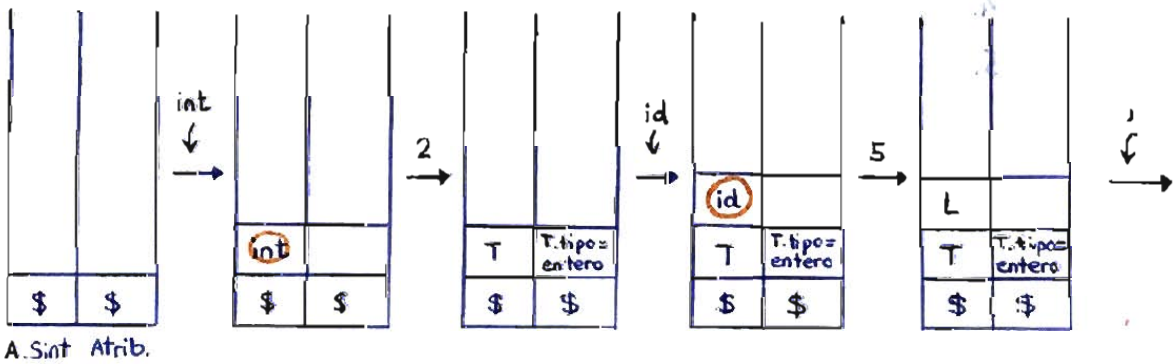
Cuando se vaya a aplicar esta regla la situación de la pila siempre será esta. L siempre va después de T.

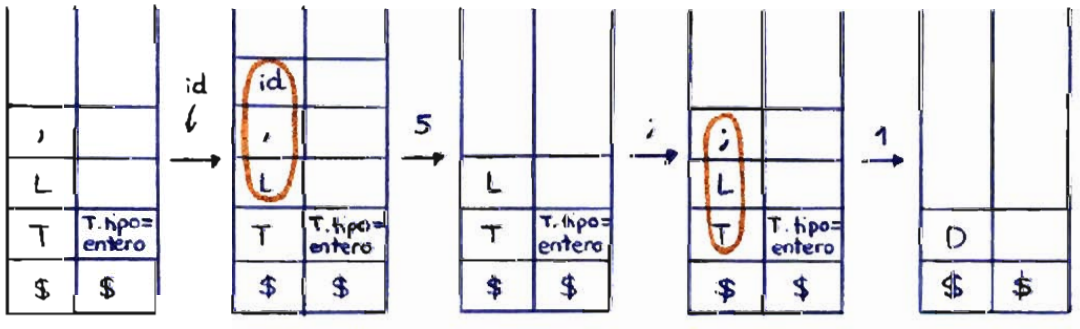


5. añadir_tipo TS (val [tope] , val [tope -1])



Ejemplo con esta gramática: Cadena de entrada: int id , id ; // int a, b;





6.4. A. SINT. DESCENDENTE TABULAR.

Evaluation of Semantic Actions in Predictive Non-Recursive Parsing

José L. Fuertes, Aurora Pérez
Dept. LSIIS
School of Computing. Technical University of Madrid
Madrid, Spain

Abstract— **To implement a syntax-directed translator, compiler designers always have the option of building a compiler that first performs a syntax analysis and then transverses the parse tree to execute the semantic actions in order. Yet it is much more efficient to perform both processes simultaneously. This avoids having to first explicitly build and afterwards transverse the parse tree, which is a time- and resource-consuming and complex process. This paper introduces an algorithm for executing semantic actions (for semantic analysis and intermediate code generation) during predictive non-recursive LL(1) parsing. The proposed method is a simple, efficient and effective method for executing this type of parser and the corresponding semantic actions jointly with the aid of no more than an auxiliary stack.**

I. Introduction

A parser uses a context-free grammar (G) to check if the input string syntax is correct. Its goal is to build the syntax tree for the analyzed input string. To do this, it applies the grammar rules. The set of valid strings generated by this grammar is the language ($L(G)$) recognized by the parser.

An LL(1) parser is built from an LL(1) grammar. The symbols of the grammar are input into a stack. The non-terminal symbol on the top of the stack and the current symbol of the input string determine the next grammar rule to be applied at any time.

A syntax-directed translator is built by defining attributes for the grammar symbols and semantic actions to compute the value of each attribute depending on others. This translator performs syntax analysis, semantic analysis, and code (intermediate or object) generation tasks.

Semantic action execution [1] can be easily integrated into several different parser types. But if you have designed a compiler with a predictive non-recursive LL(1) parser, you will find that attributes for grammar symbols that have been removed from the LL(1) parser stack are required to execute most of the semantic actions [2].

One possible solution is to build the parser tree and then transverse this tree at the same time as the semantic actions are performed. The attribute values are annotated in the tree nodes. Evidently, there is a clear efficiency problem with this solution. It also consumes an unnecessarily large quantity of resources (memory to store the whole tree, plus the node attributes, execution time…), not to mention the extra work on implementation. For this reason, a good approach is to evaluate the semantic actions at the same time as syntax analysis is performed [3] [4].

Semantic actions can be evaluated during LL parsing by extending the parser stack. The extended parser stack holds action-records for execution and data items (synthesize-records) containing the synthesized attributes for non-terminals. The inherited attributes of a non-terminal A are placed in the stack record that represents that non-terminal. On the other hand, the synthesized attributes for a non-terminal A are placed in a separate synthesize-record right underneath the record for A in the stack [5].

In this article, we introduce an algorithm for a top-down translator that provides a simpler, more efficient and effective method for executing an LL(1) parser and the corresponding semantic actions jointly with the aid of no more than an auxiliary stack.

The remainder of the article is organized as follows. Section II reviews the notions of top-down translators. Section III describes how the proposed top-down translator works, and section IV introduces an algorithm to implement this translator. Section V shows an example of how this method works. Finally, section VI outlines some conclusions.

II. Related Work

This section reviews the concepts of top-down parsers and translation schemes that can be used to cover semantic and code generation aspects.

A. Top-Down Parser

A parser applies context-free grammar rules [6] to try to find the syntax tree of the input string. A top-down parser builds this tree from the root to the leaves. At the end of the analysis, the tree root contains the grammar's start symbol and the leaves enclose the analyzed string, provided this is correct.

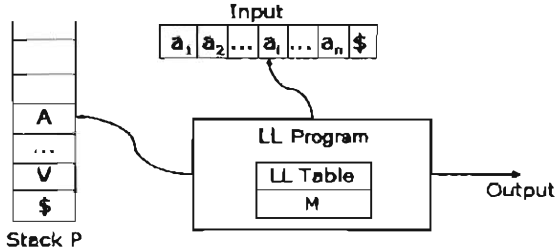


Fig. 1. Overview of an LL parser

M		a_i	...	$
A	...	A→XYZ	...	...
...	...	...	...	...

Fig. 2. LL(1) parsing table.

Additionally, a compiler parser always has to produce the same parser tree for each input string. In the case of an LL(k) parser, the mechanism used to assure that there is only one rule applicable at any time is an LL(k) grammar. This grammar finds out which rule has to be applied by looking ahead at most *k* symbols in the input string. The simplest grammar of this type is LL(1). LL(1) finds out which rule to apply by looking no further than the first symbol in the input string.

An LL parser (Fig 1) uses a stack (P) of grammar symbols and a table (M). The table (M) stores information on which rule to use to expand a non-terminal symbol on the top of the stack (A) depending on the current input symbol (a_i).

As initially configured the stack contains a symbol to indicate the bottom of stack ($) with the grammar's start symbol on top.

The rows of the LL(1) parsing table (Fig. 2) contain the non-terminal symbols. The table columns include the terminal symbols (set of valid input symbols) plus the end-of-string symbol ($). The table cells can contain a grammar production or be empty. If the parser accesses an empty cell, there is a syntax error in the input string. By definition, an LL can evidently never have more than one rule per cell.

If there is a non-terminal symbol, A, on top of the stack, the parser inspects the current input symbol, a_i, and looks up the matching table cell, $M[A, a_i]$, as shown in Fig. 1. This cell contains the rule to be applied (see Fig. 2). Then the non-terminal symbol A that was on top of stack P is removed and replaced by the right side of the applied rule. The symbol that was in the left-most position on the right side of the production (in this case X) is now on top of the stack (see Fig. 3). This is the next symbol to be expanded

If there is a terminal symbol on top of the stack, it must match the current input symbol. If they are equal, the parser takes out the symbol on top of the stack and moves ahead to the next symbol in the input string. Otherwise, the parser discovers that the syntax of the input string is incorrect.

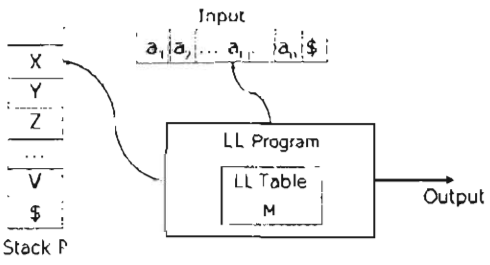


Fig. 3. Configuration of the parser after expanding rule $A \rightarrow X\,Y\,Z$.

B. *Translation Schemes*

A context-free grammar accounts for the syntax of the language that it generates but cannot cover aspects of the semantics of this language. For example, let rule (1) be:

$$S \rightarrow id := E \qquad (1)$$

Rule (1) reproduces a language's assignation sentence syntax perfectly. But it is no use for checking whether the expression and identifier types are compatible or, conversely, the programmer is trying to assign an invalid value to that identifier.

A translation scheme is a context-free grammar in which attributes are associated with the grammar symbols and semantic actions are inserted within the right sides of productions [1]. These semantic actions are enclosed between brackets { }. The attributes in each production are computed from the values of the attributes of grammar symbols involved in that production [7].

So, a translation scheme can include semantic information by defining:

- as many attributes as semantic aspects need to be stated for each symbol
- semantic actions that compute attribute values.

For rule (1), for example, the type attribute would be used for both the identifier (*id*) and the expression (E), and it would need to check that *id.type* is equal to or compatible with *E.type*.

There are two kinds of attributes: synthesized and inherited [8]. An attribute is synthesized if its value in a tree node depends exclusively on the attribute values of the child nodes. In any other case, it is an inherited attribute. In rule (2), for example, $A.s$ is synthesized and $Y.i$ is inherited.

$$A \rightarrow X\ \{Y.i := g\ (A.i,\ X.s)\}\ Y\ Z\ \{A.s := f\ (X.s,\ Y.i)\} \qquad (2)$$

An L-attributed translation scheme assures that an action never refers to an attribute that has not yet been computed. An L-attributed translation scheme uses a subset of attributes [9] formed by:

- all the synthesized attributes
- inherited attributes for which the value of an attribute in a node is computed as a function of the inherited attributes of the parent and/or attributes of the sibling nodes that are further to the left than the node.

Whereas the $Y.i$ and $A.s$ attributes in rule (2) above meet this requirement, the attribute $X.i$ would not if the rule included the semantic action $X.i := h\ (A.s,\ Z.i)$.

III. Proposed Top-Down Translator

In this section we introduce the design of the proposed top-down translator that can output the translation (the intermediate or object code in the case of a compiler) at the same time as it does predictive non-recursive parsing. This saves having to explicitly build the annotated parse tree and then transverse it to evaluate the semantic actions (perhaps also having to build the dependency graph [10] to establish the evaluation order).

We use an L-attributed translation scheme as a notation for specifying the design of the proposed translator. To simplify translator design, we consider the following criteria:

Criterion 1. A semantic action computing an inherited symbol attribute will be placed straight in front of that symbol.

Criterion 2. An action computing a synthesized attribute of a symbol will be placed at the end of the right side of the production for that symbol.

For example, (3) would be a valid rule:

$$X \rightarrow Y_1\ Y_2\ \{Y_3.i := f(X.i, Y_1.s)\}\ Y_3\ Y_4\ Y_5\ \{X.s := g(Y_3.i, Y_4.s)\} \quad (3)$$

To generate the proposed top-down translator the LL(1) parser is modified as follows. First, stack P is modified to contain not only grammar symbols but also semantic actions. Second, a new stack (*Aux*) is added. This stack will temporally store the symbols removed from stack P. Both stacks are extended to store the attribute values (semantic information).

Let us now look at how the attribute values will be positioned in each stack. To do this, suppose that we have a generic production $X \rightarrow \alpha$. This production contains semantic actions before and after each grammar symbol, where $\alpha \equiv \{1\}\ Y_1\ \{2\}\ Y_2 \ldots \{k\}\ Y_k\ \{k+1\}$.

Fig. 4 shows the parser stack P and the auxiliary stack *Aux*, both augmented to store the symbol attributes. For simplicity's sake, suppose that each grammar symbol has at most one attribute. If it had more, each position in the extended stacks would be a register with one field per attribute.

Suppose that these stacks are configured as shown in Fig. 4, with semantic action $\{i\}$ at the top of stack P. This means, as we will see from the algorithm presented in section 4, that this semantic action should be executed. There is a pointer to the top of each stack. After executing the semantic action $\{i\}$, there will be another pointer to the new top (*ntop*) of stack P.

Because of the above-mentioned Criterion 1, the semantic action $\{i\}$ uses an inherited attribute of X and/or any attribute of any symbol Y_j $(1 \le j < i)$ on the right side of the production to compute the inherited attribute of Y_i. If $i = k + 1$, the action $\{i\}$ computes the synthesized attribute of X, because of Criterion 2. The following then applies.

- *Case 1.* The semantic action $\{i\}$ computes the inherited attribute of Y_i.
  The symbol Y_i will be in stack P, right underneath action $\{i\}$, which is being executed. Thus, Y_i will be the new top (*ntop*) of stack P at the end of this execution. The reference to an inherited attribute of Y_i can be viewed as an access to stack P and, specifically, position $P\ [ntop]$.
- *Case 2.* The semantic action $\{i\}$ contains a reference to an attribute of X.
  By definition of the L-attributed translation scheme, this will always be a reference to an inherited attribute of X. Only if $i = k + 1$ will there be a reference to a synthesized attribute of X. As X will have already been removed from stack P when the rule $X \rightarrow \alpha$ was applied, the symbol X will have been entered in stack *Aux*. All the grammar symbols $Y_1, Y_2 \ldots Y_{i-1}$ (preceding the semantic action $\{i\}$) will be on top of X. These symbols will have been removed from P and inserted into *Aux*. Then any reference in $\{i\}$ to an attribute of X can be viewed as an access to stack *Aux*, specifically, position $Aux\ [top - i + 1]$.
- *Case 3.* The semantic action $\{i\}$ contains a reference to an attribute of some symbol of α.
  By definition of the L-attributed translation scheme, this attribute will necessarily belong to a symbol positioned to the left of action $\{i\}$, i.e. to one of the symbols $Y_1, Y_2 \ldots Y_{i-1}$. These symbols will have already been moved from stack P to stack *Aux*. Then any reference to an attribute of any of these symbols of α can be viewed as an access to stack *Aux* taking into account that Y_{i-1} will be on $Aux\ [top]$, Y_{i-2} will be on $Aux\ [top - 1] \ldots Y_1$ will be on $Aux\ [top - i + 2]$.

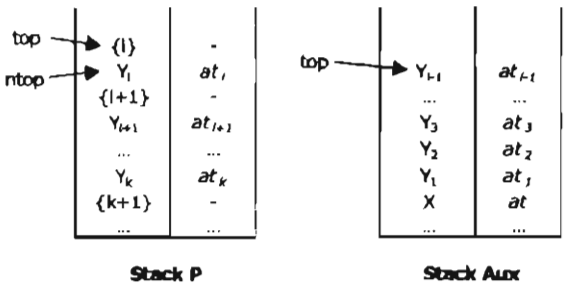


Fig. 4. Translator stacks P and *Aux* after applying $X \rightarrow \alpha$ while processing the elements of α.

The translator is implemented by programming the semantic actions and inserting them into the parser code. These semantic actions were written in terms of grammar symbols and attributes in the translation scheme. They now have to be rewritten in terms of accesses to the exact stack positions containing the values of the symbol attributes referenced in each semantic action.

The two criteria are designed merely to simplify the translator design. But the method would also work provided that the semantic action computing an inherited attribute of a symbol is located before, but not necessarily straight in front of, that symbol (Criterion 1). It would also be operational if the semantic action computing a synthesized attribute of, the symbol located on the left side of the production ($X.s$) is not at the right end of the production (Criterion 2) but depends exclusively on symbol attributes to its left. Therefore, we could also use rule (4) instead of rule (3). In the first semantic action, attribute $Y_3.i$ will be positioned in the middle of stack P, specifically $P\ [ntop - 1]$ in this case. The second semantic action is also a valid action because $X.s$ does not depend on Y_5. The referenced attributes will be in stack *Aux* ($Y_3.i$ at $Aux\ [top - 1]$, $Y_4.s$ at $Aux\ [top]$ and $X.s$ at $Aux\ [top - 4]$.

$$X \rightarrow Y_1\ \{Y_3.i := f(X.i, Y_1.s)\}\ Y_2\ Y_3\ Y_4\ \{X.s := g(Y_3.i, Y_4.s)\}\ Y_5 \quad (4)$$

IV. Top-Down Translator Algorithm

Having established the principles of the proposed top-down translator, we are now ready to present the algorithm. This algorithm is an extended version of the table-driven predictive non-recursive parsing algorithm that appears in [1].

The algorithm uses a table M'. This table is obtained from the LL parser table M by substituting the rules of the grammar G for the translation scheme rules (which include the modified semantic actions for including stack accesses instead of attributes). Then the proposed top-down translator algorithm is described as follows:

Input. An input string ω, a parsing table M for grammar G and a translation scheme for this grammar.

Output. If ω is in $L(G)$, the result of executing the translation scheme (translation to intermediate or object code); otherwise, an error indication.

Method. The process for producing the translation is:

1. Each reference in a semantic action to an attribute is changed to a reference to a position in the stack (P or Aux) containing the value of this attribute. Then the translation scheme is extended by adding a new semantic action at the end of each production. This action pops as many elements from the stack Aux as grammar symbols there are in the right side of the production. Finally, this modified translation scheme is incorporated into table M, leading to table M'.
2. Initially, the configuration of the translator is:
   - $\$S$ is in stack P, with S (the start symbol of G) on top,
   - the stack Aux is empty, and
   - $\omega\$$ is in the input, with ip pointing to its first symbol.
3. **Repeat**
   Let X be the symbol on top of stack P
   Let a be the input symbol that ip points to
   If X is a terminal **Then**
   If $X = a$ **Then**
   Pop X and its attributes out of stack P
   Push X and its attributes onto stack Aux
   Advance ip
   Else Syntax-Error ()
   If X is a non-terminal **Then**
   If $M'[X, a] = X \rightarrow \{1\}\ Y_1\ \{2\}\ Y_2 \ldots \{k\}\ Y_k\ \{k+1\}$ **Then**
   Pop X and its attributes out of stack P
   Push X and its attributes onto stack Aux
   Push $\{k+1\}, Y_k, \{k\}\ \ldots\ Y_2, \{2\}, Y_1, \{1\}$ onto stack P, with $\{1\}$ on top
   Else Syntax-Error ()
   If X is a semantic action $\{i\}$ **Then**
   Execute $\{i\}$
   Pop $\{i\}$ out of stack P
   Until $X = \$$ and $Aux = S$

V. Example

To illustrate how the method works let us use a fragment of a C/C++ grammar (Fig. 5) designed to declare local variables. Fig. 6 shows the translation scheme for this grammar.

Based on the translation scheme, apply step 1 of the algorithm described in section 4 to build the modified version of the translation scheme (see Fig. 7). In Fig. 7, references to the inherited attributes *L.type* in rule *(1)* and rule *(5)* and *R.type* in rule *(4)* have been changed to references to new top (*ntop*) of stack P. References to other attributes have been replaced with references to stack Aux. New semantic actions (calls to the *Pop* function) are included to remove symbols that are no longer needed from stack Aux. The number of symbols to be removed is the number of grammar symbols on the right side of the production. This number is passed to the *Pop* function.

Table I shows table M' for the modified translation scheme that includes references to stack positions instead of attributes. We have numbered the semantic actions with the production number and, if necessary, with a second digit showing the order of the semantic action inside the production. For instance, action *{1.1}* represents *{P [ntop]:= Aux [top]}*, the first action of production *(1)*, whereas action *{1.2}* represents *{Pop (3)}*, the second action of production *(1)*.

To illustrate how the translator works, consider the input: '*float x, y;*'. This string is tokenized by the scanner as the input string $\omega \equiv$ *float id, id;*

```
(1)  D→ T L ;
(2)  T→ int
(3)  T→ float
(4)  L→ id R
(5)  R→ , L
(6)  R→ λ
```

Fig. 5. Grammar for C/C++ variables declaration

```
(1)  D→ T {L.type:= T.type}
         L ;
(2)  T→ int {T.type:= integer}
(3)  T→ float {T.type:= float}
(4)  L→ id {insertTypeST (id.ptr, L.type);
           R.type:= L. type}
         R
(5)  R→ , {L.type:= R.type}
         L
(6)  R→ λ {}
```

Fig. 6. Translation scheme for C/C++ variables declaration.

```
(1)  D→ T {P[ntop]:= Aux[top]}
         L ; {Pop (3)}
(2)  T→ int {Aux[top-1]:= integer; Pop (1)}
(3)  T→ float {Aux[top-1]:= float; Pop (1)}
(4)  L→ id {insertTypeST (Aux[top], Aux[top-1]);
           P[ntop]:= Aux[top-1]}
         R {Pop (2)}
(5)  R→ , {P[ntop]:= Aux[top-1]}
         L {Pop (2)}
(6)  R→ λ {}
```

Fig. 7. Modified translation scheme for C/C++ variables declaration including references to stack positions

TABLE I
TABLE M' FOR THE MODIFIED TRANSLATION SCHEME ILLUSTRATED IN FIG. 7

M'	id	int	float	;	,	$
D		D → T {1.1} L ; {1.2}	D → T {1.1} L ; {1.2}			
T		T → int {2}	T → float {3}			
L	L → id {4.1} R {4.2}					
R				R → λ	R → , {5.1} L {5.2}	

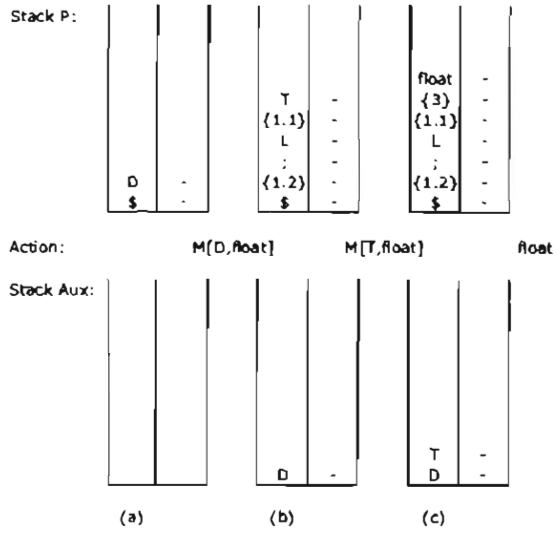


Fig. 8. First stack configurations (the input string is '*float x, y;*').

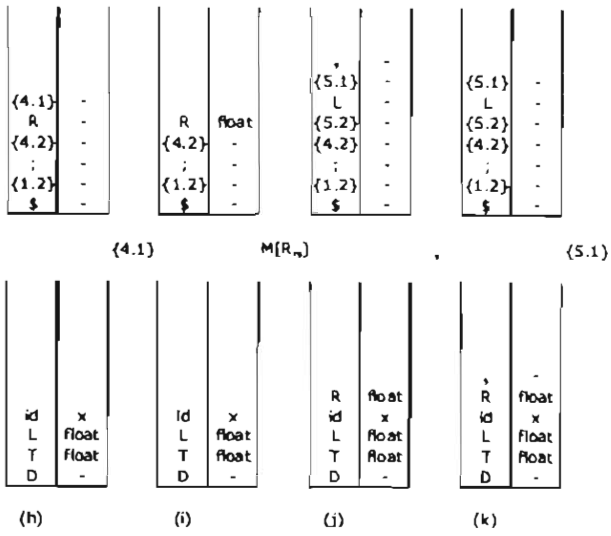


Fig. 10. Stack configurations 8 to 11 ('*, y;*' is in the input).

The stacks are initialized (Fig. 8(a)) with *$* and the grammar start symbol (*D*). Figs. 8 to 13 illustrate the different configurations of the extended stacks *P* and *Aux* (stack *P* is positioned above stack *Aux* throughout, and the action taken is stated between the two stacks).

As *D* (a non-terminal) is on top of stack *P* and *float* is the first symbol of ω, check *M [D, float]*. This gives the production $D \rightarrow T\ \{1.1\}\ L\ ;\ \{1.2\}$. Therefore, move *D* from stack *P* to stack *Aux* and push the right side of the production onto stack *P* (Fig. 8(b)).

As shown in Fig. 8(c) we find that there is a terminal *float* on top of stack *P*. As this matches the current input symbol, transfer it from stack *P* to stack *Aux* and move *ip* ahead to point to the next input symbol (*id*).

The next element on top of stack *P* is the semantic action *{3}* to be executed. This action is *{Aux [top – 1]:= float; Pop (1)}*. First, insert *float* as the attribute value of symbol *T* (the second symbol from the top of stack *Aux*). Then execute the *Pop* function, which removes one element (*float*) from the top of stack *Aux* (Fig. 9(e)).

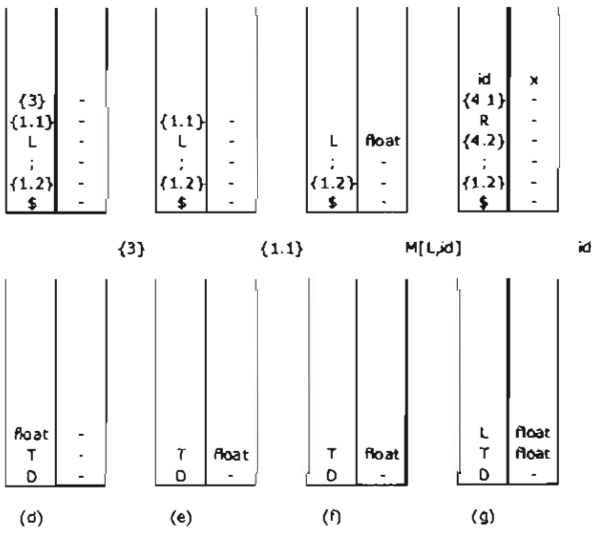


Fig. 9. Stack configurations 4 to 7 ('*x, y;*' is in the input).

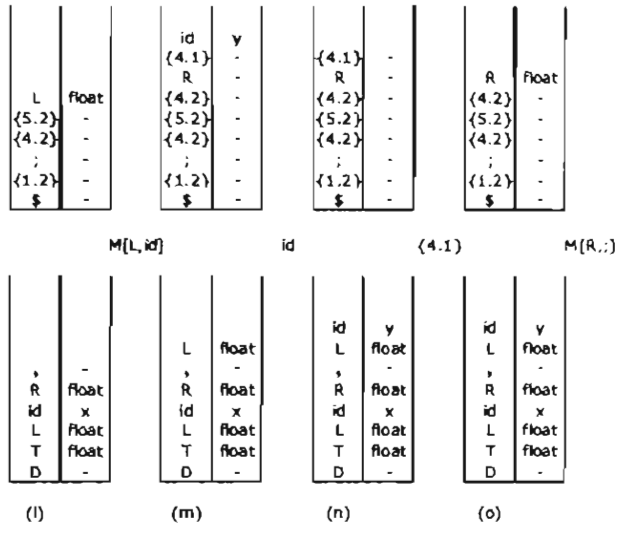


Fig. 11. Stack configurations 12 to 15 ('*y ;*' is in the input).

En todo momento tendremos disponibles en posiciones conocidas de pila todos los valores de atributos necesarios para calcular los nuevos atributos.

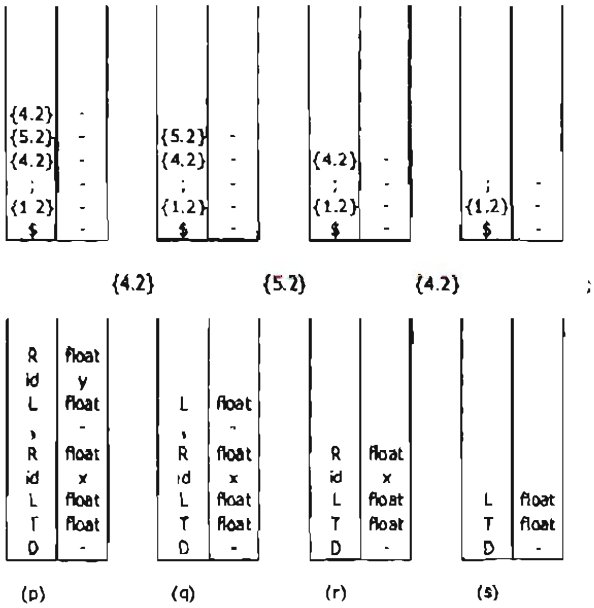


Fig. 12. Stack configurations 16 to 19 (';' is in the input).

As shown in Fig. 9(g), the *id* token has an attribute gathered directly from the scanner. This attribute is handled as a reference to the symbol table entry for this identifier. We represent the attribute using just the name of the identifier in the source code (*x*). Later the semantic action *{4.1}* is on top of stack *P* (Fig. 10(h)). Its execution copies *float* from stack *Aux* to stack *P* (*{P [ntop]:= Aux [top – 1]}*) as the attribute value of symbol *R*. The analysis continues as shown in Figs. 10 to 12.

In addition, two actions executed in this example (specifically, semantic action *{4.1}* executed in Fig. 10(h) and Fig. 11(n)) will have included type information about the *x* and *y* float identifiers in the compiler's symbol table.

Fig. 13(t) shows the execution of action *{1.2}* that removes three symbols from the stack *Aux*. Fig. 13(u) represents the algorithm exit condition.

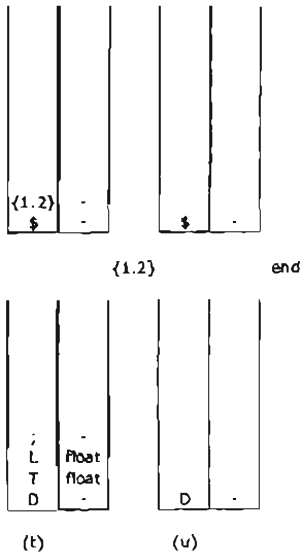


Fig. 13. Final stack configurations having read the whole input string.

VI. Conclusions

We have introduced a simple method and an algorithm to manage the evaluation of semantic actions in predictive non-recursive top-down LL(1) parsing. The main advantage of this method is that it can evaluate semantic actions at the same time as it parses. The main goal of this simultaneous execution is to save compiler resources (including both execution time and memory), since a compiler of this kind no longer needs to explicitly build a complete tree parser.

This method has been taught in a compilers course at the Technical University of Madrid's School of Computing for the last 6 years. As part of this course, students are expected to build a compiler for a subset of a programming language. About one third of students used this method, with very encouraging results. The method has proved to be easy to implement and understand.

The two criteria imposed in section 3 are merely to simplify the translator design. But the method is general and can be applied to any L-attributed translation scheme for an LL(1) grammar. Additionally, the tests run by students from our School on their compilers have shown that it is an efficient and simple way to perform the task of top-down syntax-directed translation.

References

[1] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers. Principles, Techniques and Tools*, Addison-Wesley, 1985.

[2] R. Alcker, B. Melichar, J. Tarhio, "Attribute Evaluation and Parsing", *Lecture Notes in Computer Science*, 545, Attribute Grammars, Applications and Systems, 1991, pp. 187-214.

[3] T. Noll, H. Vogler, "Top-down Parsing with Simultaneous Evaluation of Noncircular Attribute Grammars", *Fundamenta Informaticae*, 20(4), 1994, pp. 285-332.

[4] K. Müller, "Attribute-Directed Top-Down Parsing", *Lecture Notes in Computer Science*, 641, Proc. 4th International Conference on Compiler Construction, 1992, pp. 37-43.

[5] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, *Compilers. Principles, Techniques and Tools*, 2nd ed., Addison-Wesley, 2007.

[6] N. Chomsky, "Three models for the description of language", *IRE Transactions on Information Theory*, 2, 1956, pp. 113-124.

[7] T. Tokuda, Y. Watanabe, *An attribute evaluation of context-free languages*, Technical Report TR93-0036, Tokyo Institute of Technology, Graduate School of Information Science and Engineering, 1993.

[8] D. Grune, H.E. Bal, C.J.H. Jacobs, K.G. Lagendoen, *Modern Compiler Design*, John Wiley & Sons, 2000.

[9] O.G. Kakde, *Algorithms for Compiler Design*, Laxmi Publications, 2002.

[10] S.S. Muchnick, *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997.

Ejemplo: DDS y código de 3 direcciones.

Gramática:

1. S → id (lista_p) /* Llamada a procedimiento */
2. E → id (lista_p) /* Llamada a función */ → No define vectores porque van con corchetes: id [...]
3. lista_p → E
4. lista_p → E , lista_p1

Con esta gramática lista_p (lista de parámetros) puede estar formada por expresiones o funciones, es decir, la gramática define llamadas dentro del cuerpo del programa.

Por ejemplo: terminar (x, y)

Cuando llego aquí, en la Tabla de Símbolos tengo la siguiente información:

LEXEMA	TIPO	LUGAR	Nº PARAM.	TIPO_PARAM.	PASO_PARAM	PUNT_TS
terminar	n_proc		2	ent, real	ref, val	• → TS terminar

¿Cómo es el código intermedio que voy a generar?

Una llamada a una función se traduce de la siguiente manera:

- EVALUACIÓN DE LOS PARÁMETROS.
- PARAM (uno por parámetro).
- CALL

terminar (x, y) →
Param x
Param y
Call terminar; 2

terminar (a+b , y) → Código para hacer a+b /* Evaluación */
Param resultado (a+b)
Param y
Call terminar, 2

terminar (a+b, x+y) → Código para hacer a+b
Código para hacer x+y
Param resultado (a+b)
Param resultado (x+y)
Call terminar, 2

Una gramática más sencilla, donde los argumentos son sólo identificadores, sería:

1. S → id (lista_p)
2. lista_p → id
3. lista_p → id , lista_p$_1$

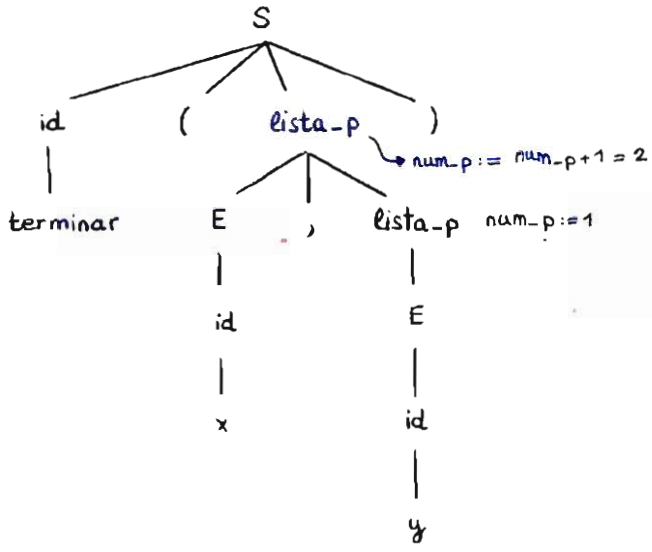
→ Usando la regla 2 y 3 construyo la lista de parámetros y calculo cuántos son.
→ Usando la regla 1 comparo y compruebo que todo sea correcto (el id, el número de parámetros y la lista obtenida). Si todo es correcto genero el código intermedio necesario.

Definición Dirigida por Sintaxis (DDS) para la gramática inicial:

```
1.  id.tipo := busca_tipo(id.entrada)        /* Está declarado */
    if (id.tipo == nombre_proc) then         /* Es un procedimiento */
        if ((busca_num_param(id.entrada) == lista_p.num_p) and
           (busca_tipo_param(id.entrada) == lista_p.tipos_param)) then
              S.tipo := tipo_ok
        else
              S.tipo := error_tipo           /* Parámetros erróneos en la llamada al procedimiento */
    if (S.tipo == tipo_ok) then
        S.codigo := lista_p.codE || lista_p.codP || Gen('Call', id.entrada, lista_p.num_p)
                    lista_p.codigo (codE = código Evaluación,
                                    codP = código Params)
    else S.tipo := error_tipo                /* Id incorrecto */
```

Nunca utilizar como etiqueta el nombre del procedimiento.

El árbol para terminar (x, y) sería :



2. E → id (lista_p)

2. if (E.tipo ≠ error_tipo) then
   E.lugar := Nuevo_Temp ();
   E.codigo := lista_p.codE || lista_p.codP || Gen (E.lugar, ':=', 'Call', busca_lugar (id.entrada))

3. lista_p → E

3. lista_p.num_p := 1
   lista_p.tipos := E.tipo
   lista_p.codE := E.codigo
   lista_p.codP := Gen ('Param', E.lugar)

Si E fuera un id (lista_p → id):
busca_tipo (id.entrada)
→ está: ok.
→ no está: fallo.

Si E fuera un id (lista_p → id):
busca_lugar (id_entrada)
→ está : ok.
→ no está : fallo.

4. lista_p → E, lista_p1

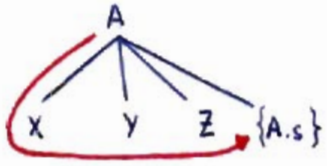
4. lista_p.num_p := lista_p1.num_p + 1
   lista_p.tipos := añadir (E.tipo, lista_p1.tipos) /* E.tipo ⊗ lista_p1.tipos */
   lista_p.codE := E.codigo || lista_p.codE
   lista_p.codP := Gen ('Param', E.lugar) || lista_p1.codP

IMPORTANTE

Dos normas para poder juntar el analizador sintáctico con la TDS sin ningún problema:

1. El valor de un atributo sintetizado de un símbolo se calcula justo después del símbolo, es decir, como último elemento del consecuente de la regla. (*)

$$A \rightarrow X\ Y\ Z\ \{A.s := g(X.x, Y.y)\}$$



2. El valor de un atributo heredado de un símbolo se calcula siempre en una acción justo delante del símbolo, es decir, inmediatamente antes del símbolo.

$$A \rightarrow X\ \{Y.R := g(A.a, X.x)\}\ Y\ Z$$

(*) Así aseguramos que ya se han calculado todos los atributos a los que hace referencia y que están disponibles en la pila de atributos.

Comprobación de tipos

Introducción

- Al proceso de comprobar si el programa fuente sigue las convenciones semánticas del lenguaje se le denomina **comprobación estática** y garantiza la detección y comunicación de algunas clases de errores de programación. Los ejemplos de comprobación estática incluyen:
  1. *Comprobación de tipos*. Un compilador debe informar de un error si se aplica un operador a un operando incompatible
     - en Pascal el operador `mod` exige operandos de tipo entero, hay que asegurarse que la "*desreferenciación*" se aplique sólo a un apuntador, que la indización se haga sólo sobre una matriz, que una función definida por el usuario se aplique al número y tipo correctos de argumentos.
  2. *Comprobación de flujo de control*.
     - en C los `break` deben aparecer en proposiciones como `for`, `while` o `switch`
  3. *Comprobaciones de unicidad*.
     - en C las etiquetas de los `case` y los `goto` deben ser únicas
  4. *Comprobaciones relacionadas con nombres*. En ocasiones, el mismo nombre debe aparecer dos o más veces. Se debe comprobar que se utilice el mismo nombre en ambos sitios
     - en JavaCC el `PARSER_BEGIN` y el `PARSER_END` deben usar el mismo nombre

Sistemas de tipos (1)

- Cada expresión tiene asociado un tipo, éste puede ser:
  - Básico: son los tipos atómicos sin estructura interna en lo que concierne al programador.
    - *boolean*, *character*, *integer*, *real*, los tipos de subrango, como `1..10` y los tipos enumerados, como `(rojo, ámbar, verde)` también pueden considerarse como tipos básicos.
  - Construido: un programador puede construir tipos a partir de tipos básicos y otros tipos construidos.
    - matrices (*array*), registros (*record*), conjuntos (*set*), los apuntadores y las funciones también pueden considerarse como tipos construidos.
- El tipo de una construcción de un lenguaje se denotará mediante una "**expresión de tipo**". De manera informal, una expresión de tipo es, o bien un tipo básico o se forma aplicando un operador llamado **constructor de tipos** a otras expresiones de tipos.

Sistemas de tipos (2)

- Un **sistemas de tipos** es una serie de reglas para asignar expresiones de tipos a las distintas partes de un programa.
- Un **comprobador de tipos** implanta un sistema de tipos. Los sistemas de tipos se pueden especificar mediante una d.d.s.
- Se dice que la **comprobación** realizada por un compilador es **estática**, mientras que la comprobación hecha al ejecutar el programa objeto se denomina **dinámica**. En principio, cualquier verificación se puede realizar dinámicamente, si el código objeto carga el tipo de un elemento junto con el valor de dicho elemento.
- Un **sistema de tipos seguro** elimina la necesidad de comprobar dinámicamente errores de tipos ya que permite determinar estáticamente que dichos errores no pueden ocurrir cuando se está ejecutando el programa objeto.
- Se dice que un **lenguaje es fuertemente tipado** si su compilador puede garantizar que los programas que acepte se ejecutarán sin errores de tipo. En la práctica, algunas comprobaciones sólo se pueden hacer dinámicamente.

106

Expresión de tipo (1)

- Una expresión de tipo se define recursivamente como:
  - Un *tipo básico* es una expresión de tipo. Un tipo básico especial, *error_tipo*, señalará un error durante la comprobación de tipos. Un tipo básico *vacío* indica "la ausencia de valor", se usa para comprobar las proposiciones.
  - Como se pueden dar *nombre* a las expresiones de tipos, el nombre de un tipo es una expresión de tipo (estos nombres se usan por ejemplo cuando se definen registros).
  - Un *constructor de tipos* aplicado a expresiones de tipos es una expresión de tipo.
  - Las expresiones de tipo pueden contener variables cuyos valores son expresiones de tipos.

Expresión de tipo (2)

- Los constructores de tipos incluyen:
  - *Matrices*. Si T es una expresión de tipo, entonces $array(I, T)$ es una expresión de tipo que indica el tipo de una matriz con elementos de tipo T y conjunto de índices I.
  - *Productos*. Si T_1 y T_2 son expresiones de tipo, entonces su producto cartesiano $T_1 \times T_2$ es una expresión de tipo.
  - *Registros*. La diferencia entre un registro y un producto es que los campos de un registro tienen nombres. El constructor de tipos record se aplicará a una tupla formada con nombres de campos y tipos de campos.
  - *Apuntadores*. Si T es una expresión de tipo, entonces $pointer(T)$ es una expresión de tipo que indica el tipo "apuntador a un objeto de tipo T".
  - *Funciones*. Se pueden considerar las funciones dentro de los lenguajes de programación como transformaciones de un *dominio tipo D* a un *rango tipo R*. La expresión de tipo $D \rightarrow R$ indicará el tipo de dicha función.

Un comprobador de tipos sencillo (1)

Gramática para lenguaje fuente

$P \rightarrow D\ ;\ E$
$D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T$
$T \rightarrow \textbf{char} \mid \textbf{integer} \mid \textbf{array}\ [\ \textbf{núm}\ ]\ \textbf{of}\ T \mid \hat{}\ T$
$E \rightarrow \textbf{literal} \mid \textbf{núm} \mid \textbf{id} \mid E\ \textbf{mod}\ E \mid E\ [\ E\ ] \mid E\ \hat{}$

La parte de un esquema de traducción que guarda el tipo de un identificador

$P \rightarrow D\ ;\ E$	
$D \rightarrow D\ ;\ D$	
$D \rightarrow \textbf{id}\ :\ T$	{ *añadetipo*(**id**.*entrada*, *T.tipo*) }
$T \rightarrow \textbf{char}$	{ *T.tipo*:=*char* }
$T \rightarrow \textbf{integer}$	{ *T.tipo*:=*integer* }
$T \rightarrow \hat{}\ T_1$	{ *T.tipo*:=*pointer*(T_1.*tipo*) }
$T \rightarrow \textbf{array}\ [\ \textbf{núm}\ ]\ \textbf{of}\ T$	{ *T.tipo*:=*array*(1..**núm**.*val*, T_1.*tipo*) }

Un comprobador de tipos sencillo (2)

Esquema de traducción para comprobar el tipo de las expresiones

$E \rightarrow \textbf{literal}$	{ *E.tipo*:=*char* }
$E \rightarrow \textbf{núm}$	{ *E.tipo*:=*integer* }
$E \rightarrow \textbf{id}$	{ *E.tipo*:=*busca*(**id**.*entrada*) }
$E \rightarrow E_1\ \textbf{mod}\ E_2$	{ *E.tipo*:=**if** E_1.*tipo*=*integer* **and** E_2.*tipo*=*integer* **then** *integer* **else** *error_tipo* }
$E \rightarrow E_1\ [\ E_2\ ]$	{ *E.tipo*:=**if** E_2.*tipo*=*integer* **and** E_1.*tipo*=*array*(*s*, *t*) **then** *t* **else** *error_tipo* }
$E \rightarrow E_1\ \hat{}$	{ *E.tipo*:=**if** E_1.*tipo*=*pointer*(*t*) **then** *t* **else** *error_tipo* }

Esquema de traducción para comprobar el tipo de las proposiciones

$S \rightarrow \textbf{id}\ :=\ E$	{ *S.tipo*:=**if id**.*tipo*=*E.tipo* **then** *vacío* **else** *error_tipo* }
$S \rightarrow \textbf{if}\ E\ \textbf{then}\ S_1$	{ *S.tipo*:=**if** *E.tipo*=*boolean* **then** S_1.*tipo* **else** *error_tipo* }
$S \rightarrow \textbf{while}\ E\ \textbf{do}\ S_1$	{ *S.tipo*:=**if** *E.tipo*=*boolean* **then** S_1.*tipo* **else** *error_tipo* }
$S \rightarrow S_1\ ;\ S_2$	{ *S.tipo*:=**if** S_1.*tipo*=*vacío* **and** S_2.*tipo*=*vacío* **then** *vacío* **else** *error_tipo* }

Esquema de traducción para comprobar el tipo de las funciones

$T \rightarrow T_1\ \text{'}\rightarrow\text{'}\ T_2$	{ *T.tipo* = T_1.*tipo*$\rightarrow$$T_2$.*tipo* }
$E \rightarrow E_1\ (E_2)$	{ *E.tipo*:=**if** E_2.*tipo*=*s* **and** E_1.*tipo*=*s*$\rightarrow$*t* **then** *t* **else** *error_tipo* }

Equivalencia de expresiones de tipos (1)

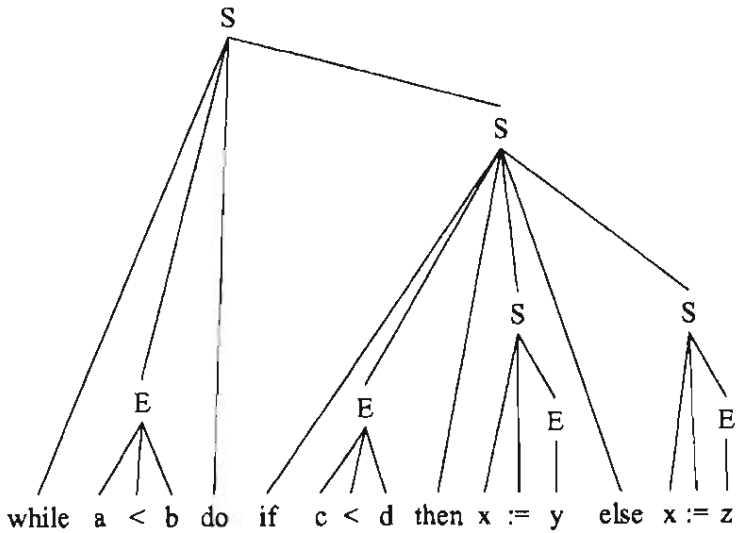
- **Equivalencia estructural**: cuando dos expresiones de tipos son, o bien el mismo tipo básico, o están formadas aplicando el mismo constructor a tipos estructuralmente equivalentes. Un algoritmo que comprueba esto es:

```
function equivest (s, t): boolean;
begin
  if s y t son el mismo tipo básico then
        return true
  else if s = array(s1, s2) and t=array(t1, t2) then
        return equivest(s1, t1) and equivest(s2, t2)
  else if s = s1×s2 and t = t1×t2 then
        return equivest(s1, t1) and equivest(s2, t2)
  else if s = pointer(s1) and t = pointer(t1) then
        return equivest(s1, t1)
  else if s = s1→s2 and t = t1→t2 then
        return equivest(s1, t1) and equivest(s2, t2)
  else
        return false
end
```

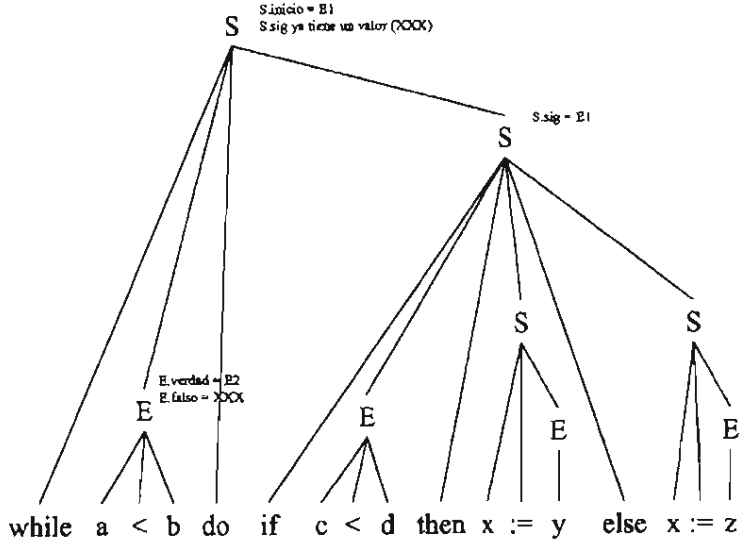

Equivalencia de expresiones de tipos (2)

- **Nombres para expresiones de tipos**
- En muchos lenguajes se puede dar nombre a los tipos.
- Para modelar esta situación, se permitirá dar nombres a las expresiones de tipo y que estos nombres aparezcan en expresiones de tipos donde previamente sólo existían tipos básicos.
- Cuando se permiten los nombres en las expresiones de tipos, surgen dos nociones de equivalencia de expresiones de tipos, según el tratamiento de los nombres:
  - La ***equivalencia de nombres*** considera cada nombre de un tipo como un tipo distinto, de modo que dos expresiones de tipo tienen equivalencia de nombre si, y sólo si, son idénticas.
  - Con la ***equivalencia estructural***, los nombres se sustituyen por las expresiones de tipos que definen, así que dos expresiones de tipos son estructuralmente equivalentes si representan dos expresiones de tipos estructuralmente equivalentes cuando todos los nombres han sido sustituidos.

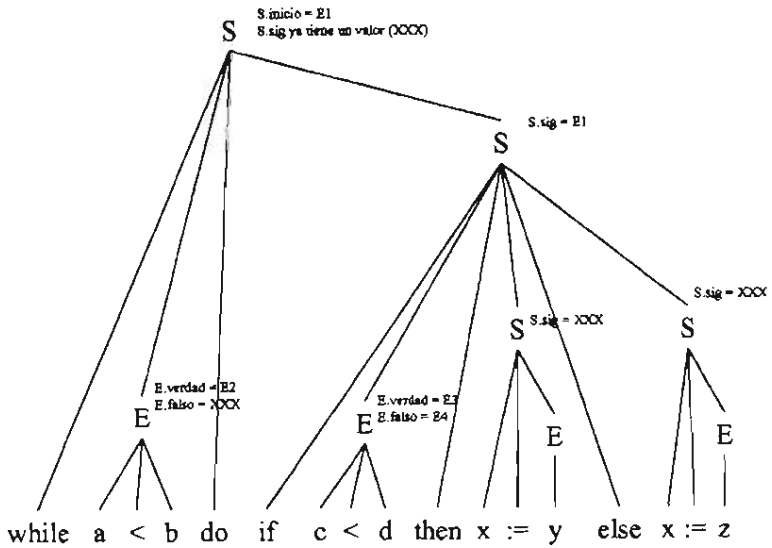
ÁRBOL WHILE POR CONTROL DE FLUJO (ÁRBOL SINTÁCTICO ANOTADO PARA LA GENERACIÓN DE CÓDIGO INTERMEDIO EN UNA INSTRUCCIÓN WHILE).



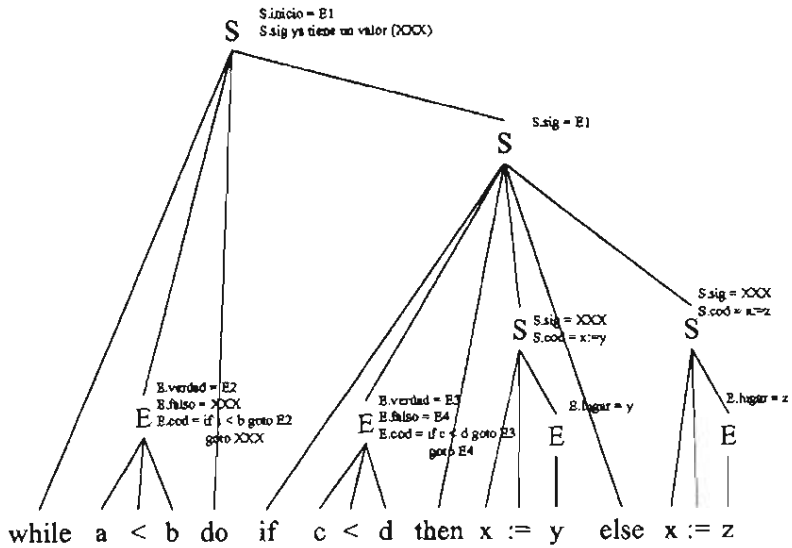
(1)



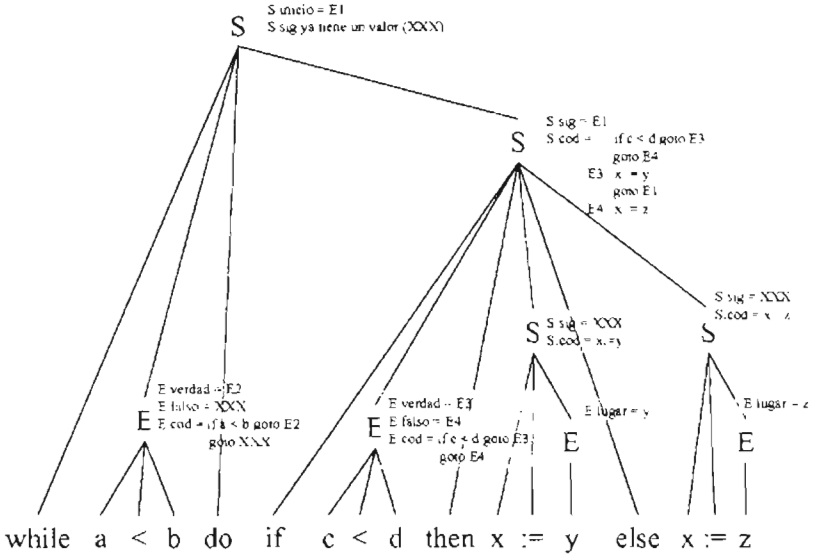
(2)



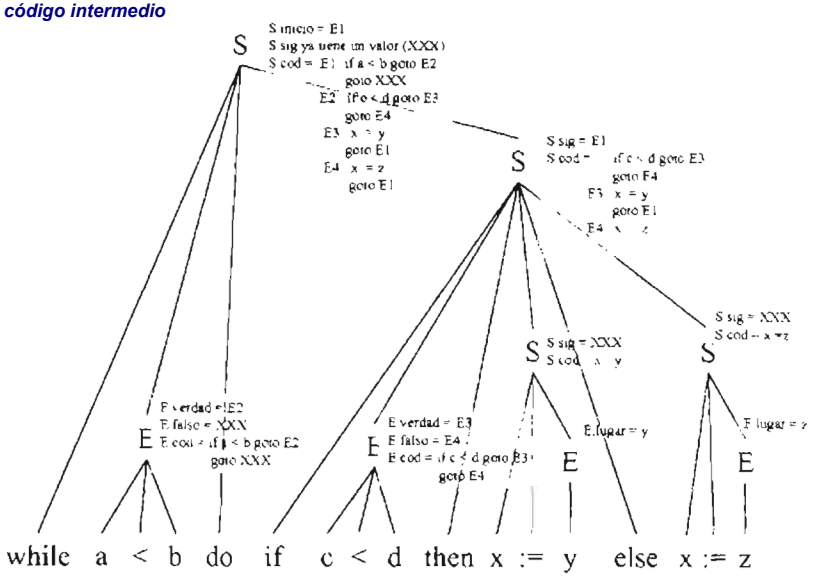
(3)



(4)


S
S inicio = E1
S sig ya tiene un valor (XXX)
S
S sig = E1
S cod = if c < d goto E3
goto E4
E3 x = y
goto E1
E4 x = z
S
S sig = XXX
S.cod = x := y
S
S sig = XXX
S.cod = x := z
E
E verdad = E2
E falso = XXX
E cod = if a < b goto E2
goto XXX
E
E verdad = E3
E falso = E4
E cod = if c < d goto E3
goto E4
E
E lugar = y
E
E lugar = z
while a < b do if c < d then x := y else x := z

(5)


S
S inicio = E1
S sig ya tiene un valor (XXX)
S cod = E1 if a < b goto E2
goto XXX
E2 if c < d goto E3
goto E4
E3 x = y
goto E1
E4 x = z
goto E1
S
S sig = E1
S cod = if c < d goto E3
goto E4
E3 x = y
goto E1
E4 x = z
S
S sig = XXX
S cod = x := y
S
S sig = XXX
S cod = x := z
E
E verdad = E2
E falso = XXX
E cod = if a < b goto E2
goto XXX
E
E verdad = E3
E falso = E4
E cod = if c < d goto E3
goto E4
E
E.lugar = y
E
E lugar = z
while a < b do if c < d then x := y else x := z

(6)

DDS para generar código de tres direcciones para la sentencia *switch*

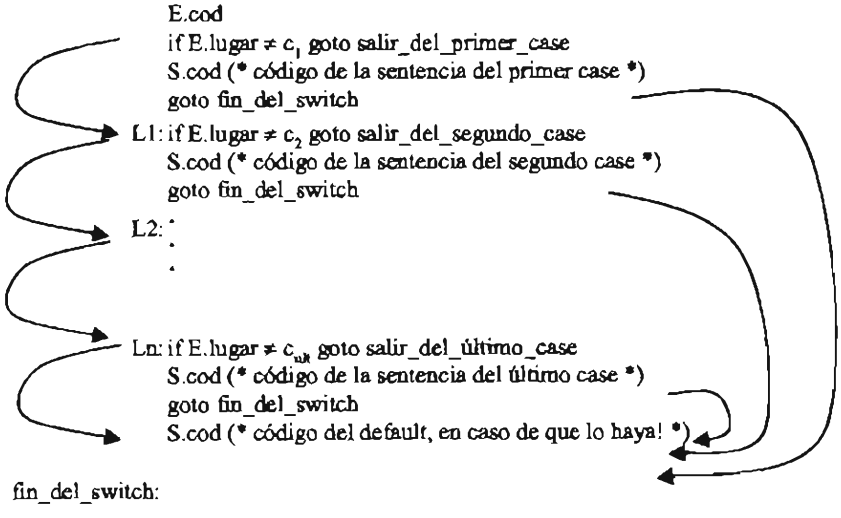
Se plantean a continuación tres soluciones diferentes para obtener la DDS pedida. La gramática para los tres casos es:

1. S → switch E begin L M end
2. L → B L_1
3. L → B
4. B → case C : S
5. M → default : S
6. M → λ
7. C → cte_entera
8. C → cte_real
9. C → cte_char

En la primera SOLUCIÓN:

- Se trabaja con el atributo S.fin (para el *switch*) al cual se da valor dentro de la propia regla (la 1), generándose la etiqueta después de que se termina de obtener todo el código del *switch* (no se trata del atributo heredado S.sig)

- La traducción que se obtiene para el *switch* evalúa la expresión E y, si ésta es distinta del valor de la primera línea *case*, salta a compararla con la de la segunda línea y así sucesivamente (se termina de una forma u otra dependiendo de que haya *default* o no). En cuanto se consigue equiparar uno de los valores *case* con el resultado de la expresión se ejecuta el código de la sentencia de ese *case* (venía secuencialmente después de la comparación) y se salta al final del *switch*.



1

Se muestran a continuación las reglas en un orden que trata de favorecer su comprensión.

4. B → case C : S

B.salir := nuevaetiqueta
B.cod := gen ('if' B.lugar ≠ C.val 'goto' B.salir) ||
S.cod || gen ('goto' B.fin) || gen (B.salir ':')

(hay que conseguir llevar el resultado de la expresión hasta B.lugar, así como el del fin del *switch* a B.fin. En la regla 1 se dispone de estos valores, que se meten en L para que L los pase a B en las reglas 2 y 3; lógicamente, L también lo pasa a "la otra L" en la regla 2)

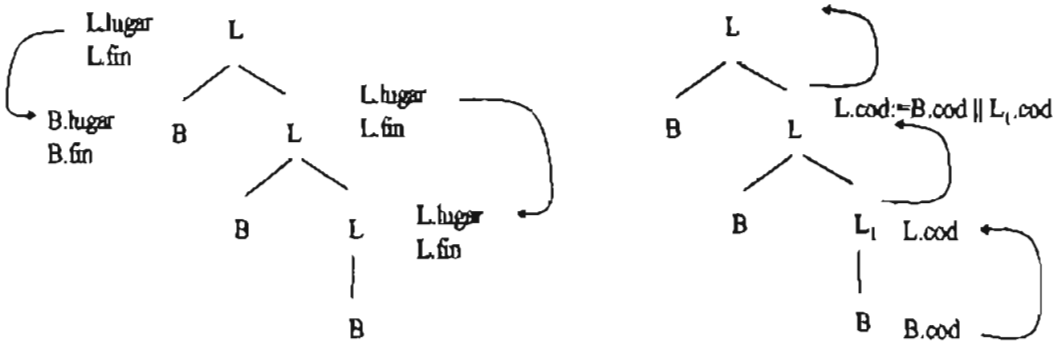
3. L → B

B.lugar := L.lugar
B.fin := L.fin
L.cod := B.cod

(el código va llevándose hacia arriba en el atributo sintetizado .cod, mientras que el resultado de la expresión y la etiqueta de fin del *switch* van bajando por el árbol llevados en atributos heredados)

2. L → B L_1

B.lugar := L.lugar
B.fin := L.fin
L.cod := B.cod || L_1.cod
L_1.lugar := L.lugar
L_1.fin := L.fin



1. S → switch E begin L M end

S.fin := nuevaetiqueta
S.cod := E.cod || L.cod || M.cod || gen (S.fin ':')
L.lugar := E.lugar
L.fin := S.fin

5. M → default : S

M.cod := S.cod

6. M → λ

M.cod := ' '

7. C → cte_entera

C.val := cte_entera.val (* lo da el A.L. *)

8. C → cte_real

C.val := cte_real.val (* lo da el A.L. *)

9. C → cte_char

C.val := cte_char.val (* lo da el A.L. *)

La segunda SOLUCIÓN presenta una pequeña diferencia: se va a trabajar con el atributo heredado .sig, cuyo valor es la etiqueta de la siguiente instrucción. La inicialización de este atributo no se ve en este ejercicio porque no corresponde a ninguna de las reglas que aquí se manejan.

1. S → switch E begin L M end	S.cod := E.cod || L.cod || M.cod L.lugar := E.lugar L.sig := S.sig M.sig := S.sig
2. L → B L_1	B.lugar := L.lugar B.sig := L.sig L.cod := B.cod || L_1.cod L_1.lugar := L.lugar L_1.sig := L.sig
3. L → B	B.lugar := L.lugar B.sig := L.sig L.cod := B.cod
4. B → case C : S	B.salir := nuevaetiqueta S.sig := B.sig B.cod := gen ('if' B.lugar ≠ C.val 'goto' B.salir) || S.cod || gen ('goto' S.sig) || gen (B.salir ':')
5. M → default : S	M.cod := S.cod S.sig := M.sig
6. M → λ	M.cod := ' '
7. C → cte_entera	C.val := cte_entera.val (* lo da el A.L. *)
8. C → cte_real	C.val := cte_real.val (* lo da el A.L. *)
9. C → cte_char	C.val := cte_char.val (* lo da el A.L. *)

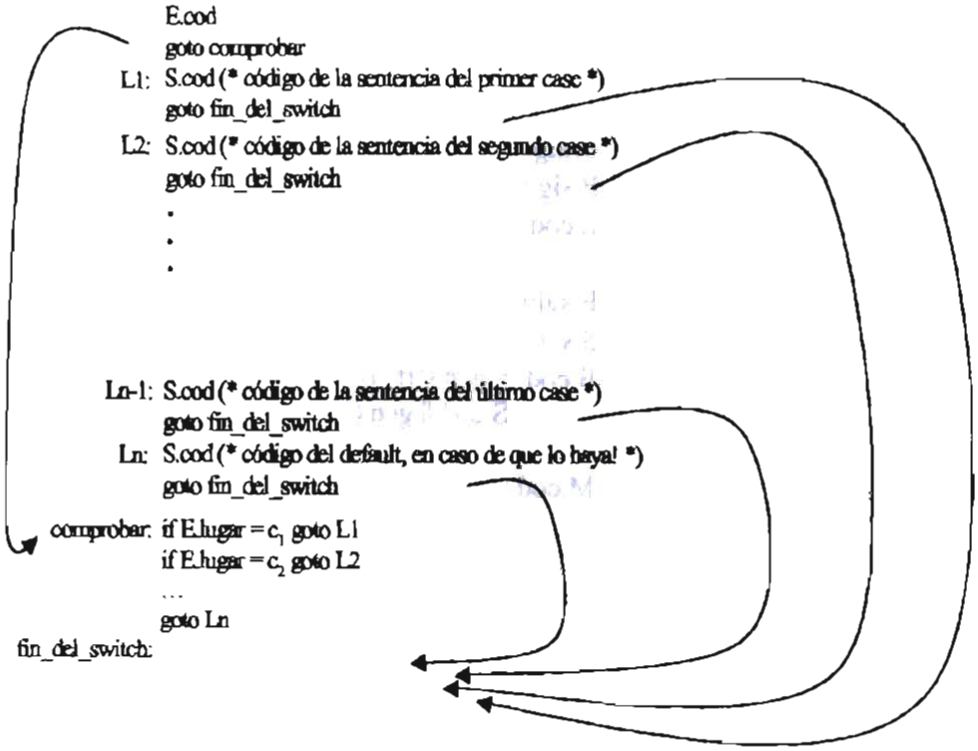
(la etiqueta de la primera instrucción que irá detrás del *switch* va bajando por el árbol mediante el atributo heredado .sig)

3

Se plantea ahora una tercera SOLUCIÓN. Esta vez:

- Se va a utilizar el atributo heredado .sig

- La traducción que se obtiene para el *switch* evalúa la expresión E y salta hacia delante a una etiqueta 'comprobar' que marca la zona donde se comprobará con qué valor de *case* coincide la E; en función de esto se saltará hacia atrás a la etiqueta correspondiente al código de la sentencia de ese *case* concreto, cuya última instrucción será un salto incondicional al final de la sentencia *switch*. Nótese que, tanto si hay *default* como si no, siempre se tienen la etiqueta "Ln" y el "goto fin_del_switch" correspondiente.



1. S → switch E begin L M end

```
S.comprob := nuevetiqueta
L.sig := S.sig
M.sig := S.sig
L.lugar := E.lugar
S.cod := E.cod || gen ('goto' S.comprob) || L.cod ||
   M.cod || gen (S.comprob ':')

for i=1 to L.long do
S.cod := S.cod ||
   gen ('if' E.lugar '=' L.casos[i] 'goto' L.etiqs[i])

S.cod := S.cod || gen ('goto' M.inicio)
```

4

Se tienen dos listas (L.casos y L.etiqs) en las que se van guardando, por cada sentencia *case*, el valor que equipara a ese *case* y la etiqueta a la que hay que saltar si es ése el que se ejecuta. Estas listas se van rellenando cada vez que se aplica la regla 2 y cuando se aplica la 3. Se rellenan con los valores de los atributo B.val y B.inicio. Con el for que se ha escrito en la acción semántica de la regla 1 se consigue que en el código 3d aparezca una instrucción de salto condicional por cada *case* que hubiera en el fuente. Al terminar este bucle, se añade al código la última instrucción que falta: un salto incondicional a la etiqueta correspondiente al código del *default*

2. $L \rightarrow B\ L_1$	B.lugar:= L.lugar B.sig := L.sig L.cod := B.cod || L_1.cod L_1.sig := L.sig L.casos := B.val + L_1.casos L.etiqs := B.inicio + L_1.etiqs L.long := L.long + 1
3. $L \rightarrow B$	B.lugar:= L.lugar B.sig := L.sig L.cod := B.cod L.casos := B.val L.etiqs := B.inicio L.long := 1
4. B → case C : S	B.inicio:= nuevaetiqueta B.cod := gen (B.inicio ':') || S.cod || gen ('goto' B.sig) S.sig := B.sig B.val := C.val
5. M → default : S	M.inicio:= nuevaetiqueta M.cod := gen (M.inicio ':') || S.cod || gen ('goto' M.sig) S.sig := M.sig
6. M → λ	M.inicio:= nuevaetiqueta M.cod := gen (M.inicio ':') || gen ('goto' M.sig)
7. C → cte_entera	C.val := cte_entera.val (* lo da el A.L. *)
8. C → cte_real	C.val := cte_real.val (* lo da el A.L. *)
9. C → cte_char	C.val := cte_char.val (* lo da el A.L. *)

5